TECHNISCHE UNIVERSITÄT ILMENAU
Institut für Technische Informatik und Ingenieurinformatik
Fakultät für Informatik und Automatisierung
Fachgebiet Softwaresysteme/Prozessinformatik
Prof. Dr.-Ing. habil. Ilka Philippow

Konzipierung und Realisierung eines Kommunikationsverfahrens zum Austausch von Modellierungsdaten und zur Erstellung und Aufrechterhaltung von Traceability zwischen verteilten Applikationen

Diplomarbeit

Zur Erlangung des akademischen Grades
„Diplom-Informatiker“

Verantwortlicher Hochschullehrer:	Prof. Dr.-Ing. habil. Ilka Philippow
Betreuer:	Dr.-Ing. Patrick Mäder
Bearbeitet von:	Keremu Abula
Matrikel:	2003
Matrikelnummer:	34615
Studiengang:	Informatik
Abgabetermin:	17. Februar 2010
Inventarisierungsnummer:	2010-02-17/027/IN03/2232

Für ***Abla Yaqup*** und ***Hawakhan Niyaz***

Atam, apam, akam, aylam we qedirlik ukilirimgha sowgha qiliman.

ئاتام، ئاپام، ئاكام، ئايلام ۋە قەدىرلىك ئۇكىلىرىمغا سوۋغا قىلىمەن.

Kurzfassung

Verteilte, miteinander vernetzte Softwaresysteme haben in den vergangenen Jahren immer mehr an Bedeutung gewonnen und durchdringen mittlerweile viele Aspekte unseres Lebens. Verteilte Systeme sind deswegen heutzutage eines der aktuellen Forschungsthemen bei der Softwareentwicklung. Da die SW-Anwendungen/Komponenten dabei häufig zwangsweise verteilt sind, wollen Benutzer auch von verschiedenen Orten aus auf gemeinsame Ressourcen, wie etwa eine Datenbank, zugreifen.
Die *TU Ilmenau* hat im Rahmen eines Forschungsprojekts ein Softwaretool/PlugIn, *TraceMaintainer*(TM) für das SW-Modellierungstool *Enterprise Architect* entwickelt. Dieses Tool kann zur Zeit nur im lokalen Rechnerbetrieb dafür eingesetzt werden, die Traceability-Links innerhalb des gesamten Entwicklungsprozesses für eine effektive, evolutionäre Entwicklung zu nutzen. Um aber den gesamten Funktionsumfang des Tools ausschöpfen zu können, muss das Tool unbedingt in einer verteilten Rechnerumgebung arbeitsfähig gemacht werden. Das Tool ist keine eigenständige SW-Anwendung und muss zur Laufzeit mit verschiedenen anderen SW-Komponenten kommunizieren. Da alle diese SW-Komponenten und das Tool selbst in Zukunft im Internet vollständig voneinander unabhängig arbeiten müssen, tritt ein neues Problem auf – die Herstellung einer Kommunikation zwischen diesen SW-Anwendungen.
Im Rahmen dieser Diplomarbeit wurde eine neue SW-Komponente für das Tool entwickelt, die die Kommunikation dieser SW-Komponenten im lokalen als auch im verteilten Rechnerbetrieb einfach unterstützt. Auf der Basis dieser grundlegenden Systemeigenschaft wurden noch eine Reihe weiterer Funktionalitäten wie *Start-Stop-Mechanismus*, *Sessionverwaltung* und *Nachrichtenverteilung* eingebaut, um die Zusammenarbeit der SW-Komponenten des Tools nach wie vor ohne Schwierigkeiten zu gewährleisten. Die neue SW-Komponente ist nach dem Prinzip von *Netzwerk-Hub* und *Hub-und-Spoke-System* realisiert und erlaubt einen eventbasierten Nachrichtenaustausch zwischen den SW-Komponenten des Tools.
Die vorliegende Diplomarbeit zeigt die Ergebnisse der Entwicklung von Programmkomponenten im Zusammenhang mit dem Entwurf eines modularen Nachrichtentransportsystems für TraceMaintainer. Durch die Vorstellung dieses Lösungsansatzes wird in dieser Diplomarbeit vermittelt, dass die Traceability-Link-Beziehungen auch in verteilten Rechnerumgebungen angewendet werden können.

Abstract

In the recent years, distributed software systems have become ever more significant and have become more important in many aspects of life. This is why distributed systems now represent one of the current fields of software development research. As software applications and their components are often necessarily distributed their users evidently wish to access shared resources, like databases, from their different locations. To enable this, setting up efficient communication between various software applications is of primary importance.
As part of a research project at *Ilmenau University of Technology*, a software tool/plugIn for the modelling tool Enterprise Architect was developed. This tool is currently used there only locally to utilize traceability links within the development process for an efficient, evolutionary development. In order to make full use of capabilities of the tool, we need to make it function in a distributed environment. The tool is no independent software application and must communicate with different other software components during operating time. As in the future all these software components and the tool itself need to be functioning on the internet completely independently, there is a new problem communication between these software applications in this new environment.
Within the scope of this thesis a new software component for the tool has been developed that supports communication between components of the tool locally as well as remotely without much difficulty. On the basis of these fundamental system features other functionalities were integrated, such as a start stop mechanism, session control and message routing. This is to ensure a continuously smooth cooperation of the tools individual software components. The new software component has been designed in accordance with the rules of *network hub* as well as *hub and spoke* and allows for an event-based exchange of messages between the software components and the tool. This thesis shows the results of program component development in relation to the design of a modular message transport system for TraceMaintainer. By presenting this approach, this thesis aims to demonstrate that traceability link relations can just as well be applied in distributed environments.

Danksagung

Diese Diplomarbeit möchte ich meinen Eltern, *Abla Yaqup* und *Hawakhan Niyaz* widmen. Nicht nur, weil sie mein Studium in Deutschland überhaupt erst ermöglichten, sondern auch, weil sie mir stets Mut und Kraft gegeben haben und mich moralisch immer unterstützt haben. Mein ganz besonderer Dank geht aber auch an meine Brüder *Keyim Abla* und *Halik Abla*, deren Bemühungen ganz entscheidend dazu beitrugen, mein Studium in Deutschland zu finanzieren.
Ich möchte auch meinen Betreuern, Frau Prof. Dr.-Ing. habil. *Ilka Philippow* und Dr.-Ing. *Patrick Mäder* für die tatkräftige Unterstützung bei der Erstellung meiner Diplomarbeit und für die hervorragende sowie intensive Betreuung danken. Vielen Dank für die hilfreichen Anregungen und die Engelsgeduld.
Ganz besonders bedanken möchte ich mich bei Frau Dr. *Eugine von Trützschler* für ihre großartige Unterstützung seit meiner Ankunft in Deutschland. Seitdem ich sie kenne, hat sie mir wie meine eigene Mutter geholfen. Sie hat stets großes Interesse an meinem Studium gezeigt und mich so gut es ging unterstützt. Sie hat mir immer Mut und Kraft gegeben. Auch war sie es, die mir in schwierigen Phasen meines Studiums ihr uneingeschränktes Vertrauen geschenkt und mit ihrer Dynamik und positiven Grundeinstellung immer wieder für den nötigen Schwung gesorgt hat. Ohne ihre beständige Unterstützung hätte ich mein Studium niemals heute hier abschließen können.
Ein herzliches Dankeschön geht an all diejenigen, die mich bei der Erstellung meiner Diplomarbeit unterstützt haben. Insbesondere bedanke ich mich bei meiner Frau *Nussrat Keram* für die Geduld und das Verständnis während der Durchführung dieser Arbeit.
Mein besonderer Dank gilt *Jochen Hoffmann* und *Claudia Bergener*. Während Claudia Bergener die erste Korrektur von Kapiteln 1 bis 3 gelesen hat, war Jochen Hoffmann aufgrund seiner langjährigen Erfahrungen nicht nur beim Korreturlesen eine große Hilfe, sondern hat mich bei der Anfertigung meiner Diplomarbeit am Ende so kräftig unterstützt. Durch seine Hilfe ist diese Arbeit wesentlich lesenswerter geworden.

Vielen Dank.

Mittwoch, den 17. Februar 2010
Ilmenau, Germany
Keram Abla

Inhaltsverzeichnis

1. Einleitung

Die Komplexität der SW-Entwicklung ist in den letzten Jahren sehr stark gestiegen und wird dies immer weiter tun [MRP06b]. Proportional dazu sind die Entwicklungsfehler bei der Umsetzung eines SW Projektmodells immer schwieriger zu erkennen und zu korrigieren. Da die steigende Komplexität von Entwicklungsprojekten und die hohe Anzahl von SW-Entwicklern, die an der Umsetzung beteiligt sind, zu schlechter Überschaubarkeit und Fehleranfälligkeit des Systementwurfs führen, ist der Aufwand zur Umsetzung der geänderten und hinzugekommenen Softwareanforderungen sehr schwer abzuschätzen. Jede Anpassung ist mit hohen Kosten und sehr viel Aufwand verbunden. Deswegen spielt Traceability in den letzten Jahren zunehmend eine wichtige Rolle bei der effektiven Entwickelung von Software [MRP06a], [Kus07].
Die TU Ilmenau hat im Rahmen eines Forschungsprojekts ein Softwarewerkzeug für das SW-Modellierungstool entwickelt, mit dem die Traceability Links innerhalb des gesamten Entwicklungsprozesses für eine effektive, evolutionäre Entwicklung genutzt werden können [Ilm06]. Das Tool wurde vom Entwicklungsteam als TraceMaintainer benannt und wird in dieser Arbeit abgekürzt als TM verwendet. Der TraceMaintainer unterstützt die halbautomatische Aktualisierungen der Traceability Beziehungen zwischen Anforderungen, Analysis und Design Modelle von Softwaresystemen, die in UML modelliert sind. Es kann die Traceability Links mit wenigen manuellen Aufwand aktualisieren [MRP07], [MGP09b].

1.1. Gegenstand der Arbeit

Wie jede andere Software, haben die SW-Entwickler den TraceMaintainer zu Beginn der Entwicklung nur ausschließlich für Rechneranlagen entwickelt und dort ausgeführt. Aktuell arbeitet der TraceMaintainer [MGP08], [MGP09b] als PlugIn von SW-Modellierungstool Enterprise Architect (EA) auf dem lokalen Desktop und erfüllt alle Systemanforderungen. Die jetzigen Aufgaben sind zumeist Berechnungen bzw. Verarbeitungen, die keinerlei Vernetzung der Rechneranlagen untereinander erfordern. Somit hat der TraceMaintainer automatisch ein sehr eingeschränktes Einsatzgebiet. Wenn der TraceMaintainer angewendet wird, kann ein Projektmodell von genau einem Anwender, aber nicht im Team bearbeitet werden. Mit diesem Tool ist es zur Zeit einem räumlich getrennten Team nicht möglich, an einem gemeinsamen Projektmodell zu arbeiten. Im Anwendungsbereich gibt es eine Vielzahl realer betriebswirtschaftlicher Prozesse, die jedoch gerade Teamarbeit erfordern und durch moderne SW-Systeme unterstützt werden sollen. Ob im Automotive-Bereich oder in der Domäne der betrieblichen Informationensysteme – überall gibt es heute verteilte, interagierende Softwaresysteme, manchmal sogar als unvermeidbarer Teil der Softwareentwicklung. Reine lokale Anwendungen wie TraceMaintainer können diese Ansprüche nicht erfüllen. Deswegen muss der TraceMaintainer in einer verteilten Umgebung arbeitsfähig sein.

1.2. Ziel der Arbeit

Ein verteiltes System ist ein System, in dem sich HW- und SW-Komponenten auf vernetzten, räumlich verteilten Rechnern befinden und miteinander durch den Austausch von Nachrichten kommunizieren [KB07]. Da die SW-Anwendungen bzw. SW-Komponenten dabei häufig zwangsweise verteilt sind, wollen die Benutzer auch an verschiedenen Orten auf gemeinsamen Ressourcen, wie etwa eine Datenbank, zugreifen [Ill07]. Dazu ist die Herstellung einer effizienten Kommunikation zwischen beliebigen SW-Anwendungen sehr wichtig. Das ist gerade bei dem TraceMaintainer der Fall. Der TM muss zur Laufzeit mit verschiedenen SW-Anwendungen bzw. SW-Komponenten kommunizieren. Effiziente und modernere Kommunikationsverfahren sind nötig, die den SW-Komponenten des TraceMaintainer diese Kommunikation ermöglicht.
Die vorliegende Arbeit beschreibt einen Ansatz zur Realisierung eines speziellen Nachrichtentransportsystems, das die räumlich verteilten SW-Komponenten vom TM bei ihrer Kommunikation unterstützt. Im Rahmen dieser Arbeit wurde das System RemotingHub benannt. Wie der Name schon sagt, sollte das System als eine Art von Hub die Nachrichtenpakete von einer Komponente zur anderen bzw. zu einer bestimmten Komponente verteilen können. Um eine effiziente Kommunikation zwischen den SW-Komponenten herzustellen, musste zunächst der aktuelle Stand der Technik bekannt sein. Im Rahmen dieser Diplomarbeit wird zunächst ein Überblick über die zugrunde liegenden aktuellen Technologien gegeben, die unter verschiedenen SW-Komponenten eine problemlos funktionierende Kommunikation im Netzwerk und lokal auf dem PC herstellen. Den Schluss bildet die Anwendung des entwickelten Konzeptes und deren Implementierung in das System mit Hilfe der passenden Technologie.
Das allgemeine Ziel dieser Arbeit ist, einen möglichen Lösungsweg für das Nachrichtentransportsystem zu konzipieren und zu realisieren, durch den die verteilten SW-Anwendungen bzw. SW-Komponenten von verschiedenen Orten auf gemeinsame Ressourcen, wie etwa ein TraceMaintainer, zugreifen können, damit die Kommunikation zwischen den SW-Komponenten des TraceMaintainers nicht nur auf dem lokalen Rechner, sondern auch über Rechnergrenzen hinweg einfach unterstützt werden kann.Darauf aufbauend wird ein Ansatz entwickelt, nach dem sich bestimmte SW-Komponenten von TM starten und schließen lassen. Dieses Konzept soll erweiterbar und offen für weitere Anpassungen gehalten werden. Die prototypische Implementation des Systems soll eine eigenständige Komponente des TraceMaintainer darstellen, so dass am Ende jede Art von Client daran angeschlossen werden kann. Da das hier zu realisierende System nur einen Knotenpunkt der Kommunikation zwischen verschiedenen Kommunikationspartnern darstellt, macht es keinen Sinn, wenn zunächst das System allein entwickelt und zum Schluss in einer Testumgebung nach den Funktionalitäten, der Anwendbarkeit, der Zuverlässigkeit, usw. getestet wird. Deswegen soll es komplett in einer Testumgebung integriert entwickelt werden.

1.3. Aufbau der Arbeit

Nach einer kurzen Einleitung wird in Kapitel 2 zunächst auf die notwendigen theoretischen Grundlagen der verteilten Anwendungen, der Kommunikationsformen sowie der Architekturmodelle zur Realisierung der Kommunikation eingegangen. Es dient dazu, den Lesern wichtige

grundlegende Informationen zu dieser Arbeit geben zu können. Außerdem werden in diesem Kapitel noch relevante Konzepte und Produkte vorgestellt.
Kapitel 3 umfasst den Stand der Technologie, welche die moderne Kommunikation zwischen den verteilten Systemen unterstützt. Zunächst wird eine kurze Übersicht über eine Reihe von ausgewählter Technologie, die möglicherweise bei der Realisierung der Zielsetzung dieser Arbeit zum Einsatz kommen könnte, gegeben und danach jede Einzelne kurz und knapp bezüglich des Themas vorgestellt. Zum Schluss werden die Technologien, die am ehesten für die Realisierung ausgewählt werden können, gegenüber gestellt. Das Hauptaugenmerk liegt dabei auf den .Net-Technologien. Der Grund dafür ist der TraceMaintainer, der mit Hilfe von .Net Technologie entwickelt worden ist.
In Kapitel 4 wird der Aufbau des TraceMaintainer dargestellt und kurz auf die SW-Komponenten des Systems eingegangen. Den Hauptteil der Arbeit stellen die Kapitel 5 und 6 dar. In diesen wird die Entwicklung des Ansatzes und die Konzeption des Nachrichtentransportsystems beschrieben. Beginnend mit Untersuchungen verschiedener Ansätze zu Problemstellungen wird eine Lösungsidee des hier entwickelten Ansatzes präsentiert. Dem folgt eine detaillierte Beschreibung der Umsetzung sowie die Verarbeitung der eingehenden Nachrichten und deren angemessene Verteilung an Abonnenten, an die Verwaltung der SW-Komponenten, usw. Die Implementierung eines Prototyps des konzipierten Systems wird in Kapitel 6 vorgestellt. Das Kapitel 7 widmet sich dem Test des hier realisierten Prototyps des Systems; hier werden hauptsächlich die Testmethode und die nötigen Komponenten der Testumgebung vorgestellt. Am Schluss der Arbeit erfolgt eine Zusammenfassung und die Bewertung des Erreichten sowie ein Ausblick zu möglichen Weiterentwicklungen.

2. Theoretische Grundlagen

Bevor mit der Zielsetzung dieser Arbeit begonnen wird, müssen einige theoretische Grundlagen gesetzt sein. Dazu zählen unter anderem verteilte Anwendungen, die Kommunikationsmodelle und einzusetzende SW-Technologien. Hinzu kommen noch die relevanten Konzepte und Produkte zu dem hier zu realisierenden System.

2.1. Verteilte Anwendungen und ihre Eigenschaften

Durch die Realisierung des Nachrichtentransportsystems wird es möglich sein, das UML-Modellierungstool und den TraceMaintainer räumlich zu trennen. Somit kann der TraceMaintainer von beliebigen Rechnern in einem Netzwerk erreicht und angewendet werden. Dabei müssen die Nachrichten zwischen eigenem lokalen PC und einem entfernten PC übertragen werden. In der Informationstechnik spricht man in diesem Fall von einer verteilten Anwendung. Um den Begriff *Verteilte Anwendung* zu erläutern, muss zunächst ein weiterer Begriff, der Begriff *Verteiltes System*, definiert werden. In [Ben02] wird der Begriff wie folgt definiert:

> „Ein verteiltes System ist definiert durch eine Menge von Funktionseinheiten oder Komponenten, die in Beziehung zueinander stehen (Client-Server-Beziehung) und eine Funktion erbringen, die durch die Komponente alleine nicht erbracht werden kann."

In [TS03] werden aus einer ähnlichen Definition zwei Aspekte abgeleitet. Der erste Aspekt bezieht sich auf die Hardware. Anhand der genannten Definition kann ein verteiltes System auf mehrere autonome Rechner verteilt sein. Der nächste Aspekt ist der Benutzer. Benutzer eines verteilten Systems haben demnach den Eindruck, als würden sie es mit einem einzigen System zu tun haben. Das dahinter liegende System ist für den Benutzer völlig unsichtbar bzw. transparent. Die Anwendung scheint für den Nutzer lediglich aus dem Rechner zu bestehen, an dem er seine Eingaben tätigt. Diese beiden Aspekte machen die Ziele [Ben02], [TS03], [Ill07] verteilter Systeme deutlich.

- Offenheit: Mithilfe geeigneter Schnittstellen kann ein verteiltes System seine Dienste anderen Anwendungen zur Verfügung stellen. Somit ist es möglich, aus sehr unterschiedlichen Anwendungen heraus auf bereitgestellte Dienste zuzugreifen, vorausgesetzt, die jeweilige Anwendung implementiert die notwendigen Schnittstellen.
- Skalierbarkeit: Verteilte Systeme sollen leicht erweiterbar sein. Dies betrifft sowohl das Hinzufügen weiterer Hardware als auch die Erweiterung des Funktionsumfangs angebotener Dienste. Nachdem nun ein verteiltes System definiert und seine Ziele erläutert wurden, kann eine Verteilte Anwendung als eine Software beschrieben werden, die aus

mehreren Komponenten besteht. Die einzelnen Teile der gesamten Anwendung verteilen sich auf unterschiedliche Rechner.

- Unsichtbarkeit: Es ist die wichtigste Eigenschaft von verteilten Systemen und in der englischsprachigen Literatur unter dem Begriff Transparenz bekannt [Ben02]. Nach dieser Eigenschaft soll ein verteiltes System für den Anwender unsichtbar sein. Der Anwender soll im Idealfall nicht bemerken, dass die Anwendung, mit der er gerade arbeitet, auf mehreren Rechnern verteilt ausgeführt wird. Ein verteiltes System stellt sich also für den Bediener wie ein einziger Rechner dar.
- Den Zugang zu entfernten oder gemeinsam genutzten Ressourcen: Im hier diskutierten speziellen Fall von RemotingHub können verteilte Systeme dazu beitragen, entfernte Ressourcen zugänglich zu machen. Es ist dem Entwickler, der auf einem Modellierungstool arbeitet, somit möglich, einen entfernt aufgestellten TraceMaintainer über Rechnergrenzen hinweg zu benutzen.

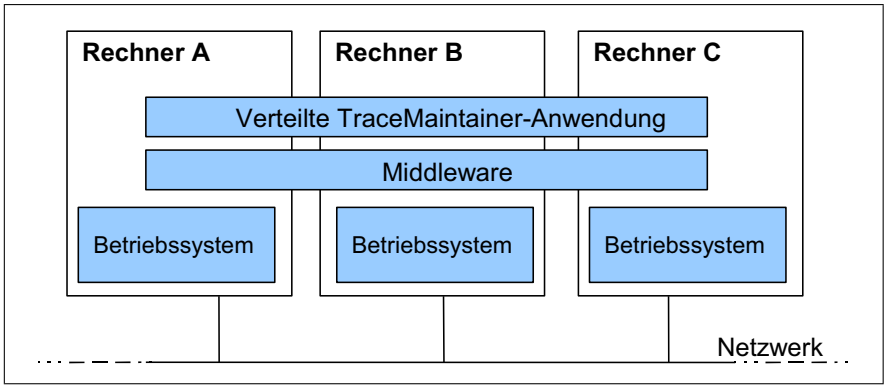

Abbildung 2.1.: Verteilte Anwendung und Middleware

Es gibt verschiedene Möglichkeiten, eine verteilte Anwendung zu implementieren. In der Abbildung 2.1 wurden die Schichten verteilter Systeme verdeutlicht. Wie zu erkennen ist, kann durch geeignete Middleware [TS03], [KCH04] realisiert werden, die Anwendung auf Rechner mit unterschiedlichen Betriebssystemen zu verteilen. Lösungen, die in der Programierung weitaus komfortabler sind, beruhen auf Middleware-Ansätzen. Middleware liegt, wie der Name schon sagt, in der Architektur zwischen dem Netz und der Anwendung. Ihre wesentliche Aufgabe besteht darin, die Komplexität des Netzes vor der Anwendung bzw. dem Programierer zu verbergen [Ben02], [EF03]. Zu fragen ist jetzt, wie die einzelnen Teile einer verteilten TraceMaintainer-Anwendung in eine logische Beziehung zueinander gebracht werden. Um diese Fragestellung zu behandeln, werden im folgenden Abschnitt einige grundlegende Konzepte vorgestellt.

2.2. Systemmodelle: Die Architektur verteilter Systeme

Während der Blütezeit der prozeduralen Programmierung waren monolithische Architekturen lange Zeit das dominante Strukturierungsmittel [Ill07]. Mit zunehmendem Einsatz der Objektorientierung haben sie jedoch ihre Attraktivität bei der Realisierung verteilter Systeme verloren, weil sich neue, komplexere Architekturstrukturen sowie z.B. Service-orientierte Architekturen (SOA), Peer-to-Peer-Architekturen, Client-Server-Architekturen herausgebildet haben [Ill07], [Ben02]. Die Programmierung der verteilten Systeme hängt von den Anforderungen ab, sodass eine Applikation auf der Basis welcher von diesen Architekturen entworfen und implementiert werden soll. Die meisten verteilten Anwendungen stützen sich heute auf das Client-Server-Modell, in dem ein Serverprozess oder ggf. mehrere Serverprozesse auf Anforderungen von Clients warten und diese beantworten. Ein anderes Modell ist die sog. Peer to peer Kommunikation, in der zwei Anwendungsprozesse gleichberechtigt miteinander kommunizieren [MBW08]. Die SOA wird für die komplexeren Gesamtanwendungen bevorzugt eingesetzt [Ill07], [KCH04]. Diese Arbeit beschränkt sich auf die Client-Server-Modelle und Peer-to-Peer Modelle.

2.2.1. Das Client-Server-Modell

Der bedeutendste Trend bei Informationssystemen in den letzten Jahren ist das Aufkommen des Client-Server-Modells. Dieses am weitesten verbreitete Modell [Sch93], [MBW08] für verteilte Anwendungen bindet lokale und isolierte Rechner zusammen und fördert ihre Zusammenarbeit. Beide Begriffe wurden bereits in der Definition des verteilten Systems angedeutet. Tanenbaum definiert beide Begriffe in [TS03] wie folgt:

> „Ein Server ist ein Prozess, der einen bestimmten Dienst implementiert, beispielsweise einen Dateisystemdienst oder einen Datenbankdienst. Ein Client ist ein Prozess, der einen Dienst von einem Server anfordert, indem er eine Anforderung sendet und dann auf die Antwort des Servers wartet.“

Zusammen bilden sie ein komplettes System mit unterschiedlichen Bereichen der Zuständigkeit, wobei diese Zuständigkeiten oder die Rollen fest zugeordnet sind: Entweder ist ein Prozess ein Client oder ein Server. Clients und Server können auf dem gleichen oder auf unterschiedlichen Rechnern ablaufen [Ker08]. Wie in Abbildung 2.2a ersichtlich, fordert der Clientprozess eine Operation oder einen Service von einem anderen Serverprozess an. Nach Erhalt einer Anforderungsnachricht führt der Server den angeforderten Service aus und gibt dem Client ein Resultat oder das Ergebnis des Services zurück. Dieses einfache Client-Server-Modell führt zu einer Reduktion auf mehrere Clients und einem Server und es legt fest, wie eine Anwendung den Service eines Servers in Anspruch nehmen kann [Ben02].
Wie in Abbildung 2.2b dargestellt, besteht ein Client-Server-System aus zwei logischen Teilen:

1. Ein oder mehrere Clients, die die Services oder Daten der Server in Anspruch nehmen und somit anfordern.
2. Ein Server, der Services und Daten zur Verfügung stellt.

Clients und Server sind zwei Ausführungseinheiten mit einer Konsumenten- und Produzentenbeziehung. Clients dienen als Konsumenten und tätigen Anfragen an Server für Services und

benutzen dann die Rückantworten zu ihrem eigenen Zweck und ihre Aufgabe. Server spielen die Rolle des Produzenten und führen die Daten- bzw. Serviceanfragen aus, die von den Clients gestellt worden sind [Ker08], [Sch93].
Der Ablauf der Interaktion, die zwischen zwei Rechnern stattfindet, soll die gleiche Semantik besitzen, d.h. lokale und entfernte Interaktion solle die gleiche Syntax und Semantik besitzen. Selbst wenn die Anforderungen oder Aufrufe der Clients keinerlei syntaktische Unterschiede zwischen lokaler und entfernter Interaktion aufweisen, so müssen doch die semantischen Unterschiede in eine die Interaktion benutzende Anwendung einfließen [Ben02].

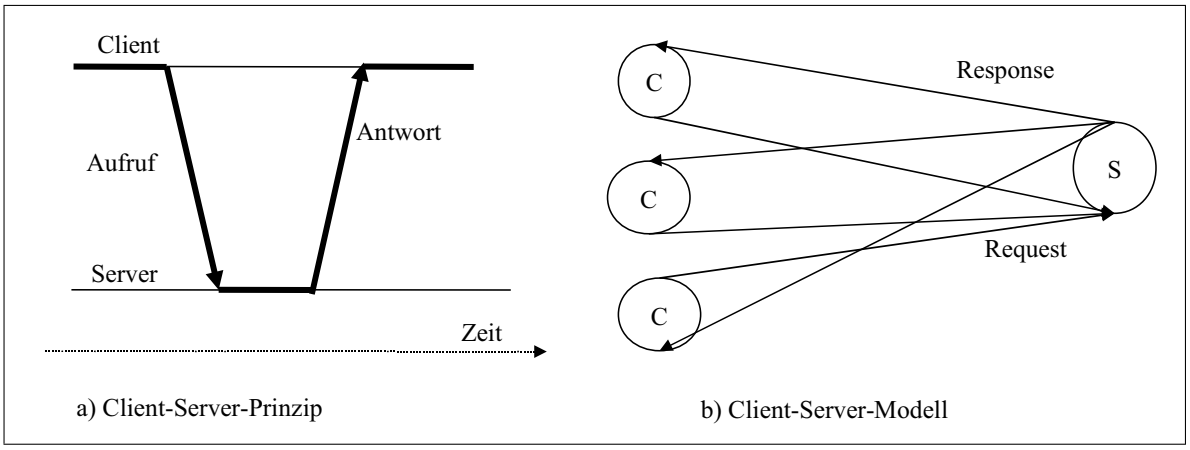

Abbildung 2.2.: Client-Server-Modelle

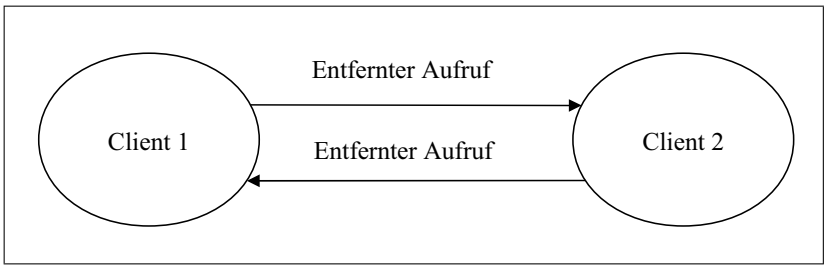

Abbildung 2.3.: Das Peer-To-Peer-Prinzip

2.2.2. Peer-to-Peer-Modelle

Peer-to-Peer-Modelle(PTP-Modelle) verwenden einen zur CS-Architektur gegensätzlichen Ansatz zur Implementierung eines verteilten Systems. Ein PTP-System ist ein dezentralisiertes, verteiltes System, welches aus gleichberechtigten Knoten besteht, *Peers* genannt. Die Eigenschaft, dass die Daten nicht auf zentralen Servern liegen, sondern direkt auf den einzelnen

verbundenen Rechnern ist der entscheidende Vorteil gegenüber dem Client-Server-Modell. Das System organisiert sich selbst und hat die Fähigkeit, sich an Veränderungen, wie z.B. Fehler, anzupassen. Wie in Abbildung 2.3 gezeigt, kann jeder Peer von anderen einen entfernten Dienst anfordern und auch selbst einen Dienst anbieten. So gesehen ist die PTP-Architektur das Gegenteil der Client-Server-Architektur [Ill07]. Sie werden heutzutage im Kontext von File-Sharing-Anwendungen im Internet und die Internet-Kommunikation wie Skype eingesetzt.
Aus Abbildung 2.4 werden topologische Unterschiede beider Modelle ersichtlich. Der Gegensatz zum PTP-Modell ist das CS-Modell. Während im CS-Modell ein Server einen Dienst anbietet und ein Client diesen Dienst nutzt, ist im PTP-Netz diese Rollenverteilung aufgehoben [MNW03]. Jeder Teilnehmer ist ein Peer, denn er kann gleichzeitig einen Dienst nutzen und selbst anbieten. Während man mit dem CS-Modell eine sternförmige Netzwerkkommunikation aufbaut, die nachteilig ist, da alle Clients von einem oder einigen Servern abhängig sein können, kann man mit PTP-Modellen diese Zentralisierung vermeiden. Ein PTP-Modell ist daher vorteilhaft, wenn „Zentrale“ Funktionen der Server sowie diese eines Client-Server-Modells vermieden wird, weil dann keine Einrichtung im PTP-Modell vorhanden ist, von der alle anderen abhängig sind. Eine noch weitere wichtige Eigenschaft ist die Robustheit der PTP-Systeme. Obwohl die Verbindungen innerhalb des Systems immer wieder unterbrochen werden, wenn ein sogenannter Peer das System verlässt, korrigiert die Robustheit eines PTP-Systems umgehend diesen Zustand, so dass das System dadurch nicht behindert wird. Dadurch, dass jeder Peer alles anbieten kann und alles von jedem anderen Peer nutzen kann, muss auf jedem Peer festgehalten werden, wer welche Ressourcen(Drucker, Dateien, etc.) nutzen darf. Dies führt zu einem hohen Verwaltungsaufwand auf jedem Rechner [Ben02].

2.3. Kommunikationsformen

viele verteilten Systeme und Anwendungen setzen direkt auf dem einfachen nachrichtenorientierten Modell auf, das die Transportebene unterstürtzt [TS03]. Die Interaktion zwischen Server und Client, wie bereits im vorhergehenden Abschnitt 2.2.1 beschrieben, könnte in der blockierenden(Synchroner Fall) oder nicht-blockierenden(Asynchroner Fall) Form koordiniert werden. Weiterhin wird grundsätzlich zwischen meldungsorientierter und auftragsorientierter Kommunikation unterschieden. In diesem Abschnitt werden diese vier grundlegenden Kommunikationsformen vorgestellt [MBW08].

2.3.1. Synchrone und Asynchrone Kommunikation

Beim Nachrichtenaustausch unterscheidet man nach [Web98] grundsätzlich zwei Formen der Kommunikation: Synchrone und Asynchrone Kommunikation. Wenn der Client nach Absenden der Anforderung an den Server auf eine Rückantwort wartet, bevor er anderen Aktivitäten nachgeht, so liegt der blockierende oder synchrone Fall vor. Obwohl dieses Vorgehen sich leicht implementieren lässt, ist jedoch ineffizient in der Ausnutzung der Prozessorfähigkeiten des Clients. Dagegen sendet im asynchronen Fall der Client nur seine Anforderung und arbeitet sofort weiter. Deswegen wird in diesem Fall von einer nicht blockierenden oder asynchronen Interaktion gesprochen. Irgendwann später nimmt der Client dann die Rückantwort entgegen. Der

Vorteil dieses Verfahrens ist, dass der Client parallel zur Nachrichtenübertragung weiterarbeiten kann und den Clientprozess nicht durch aktives Warten belastet, wie beim blockierenden Fall. Jedoch muss bei dieser Methode der erhöhte Effizienzgewinn mit erhöhter Kontrollkomplexität bei Erhalt der Rückantwort erkauft werden [Ben02].

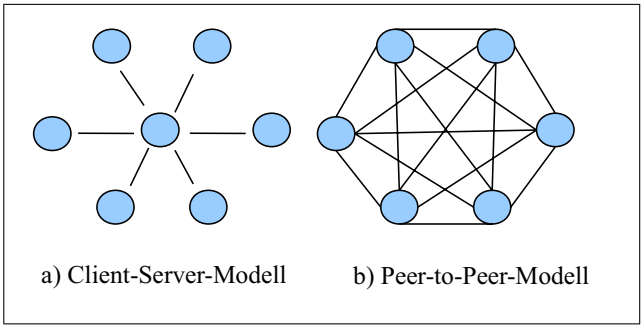

Abbildung 2.4.: Ein topologischer Vergleich

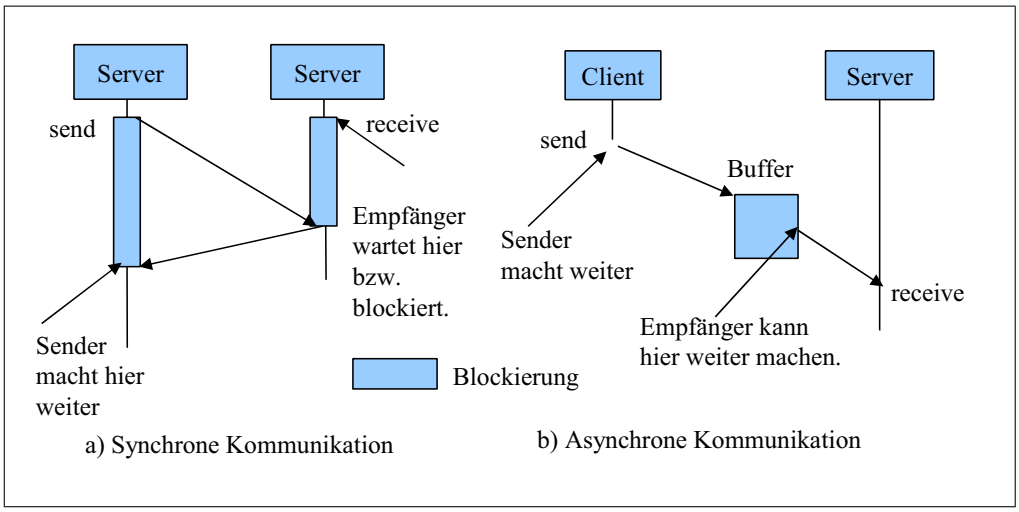

Abbildung 2.5.: Synchrone und Asynchrone Kommunikation

in jedem Fall werden die Puffer für ankommende Nachrichten in den Protokollinstanzen (meist im Betriebssystemkern) verwaltet [MBW08], [Ben02]. Vorteile der synchronen Kommunikation (in Abbildung 2.5a) sind die automatische Synchronisation zwischen Sender und Empfänger sowie das Einsparen von Pufferspeicher. Nachteilig sind, die eingeschränkte Parallelität und das ggf. lange blockieren, wenn der Empfänger keine Zeit hat. Aber auch der Empfänger selbst blockiert, wenn er auf Anfragen von Clients wartet. Bei der asynchronen Kommunikation, wie sie in der Abbildung in 2.5b skizziert ist, besteht der Vorteil z.B. in der zeitlichen Entkopplung von Sender und Empfänger. Damit kann auch eine bessere Parallelität sowie eine ereignisgesteuerte

Kommunikation ermöglicht werden. Als Nachteil zu nennen ist, dass eine Zwischenpufferung der Nachrichten notwendig ist. Wenn der Puffer voll wird, dann führt dies trotzdem zum Blockieren, um eine gesicherte Übertragung zu gewährleisten [MBW08].

	synchron	asynchron
Meldungsorientiert	Datagramm	Rendezvous
Auftragsorientiert	Asynchroner entfernter Diesntaufruf	Synchroner entfernter Diesntaufruf

Tabelle 2.1.: Synchrone und Asynchrone Kommunikation (nach [Web98])

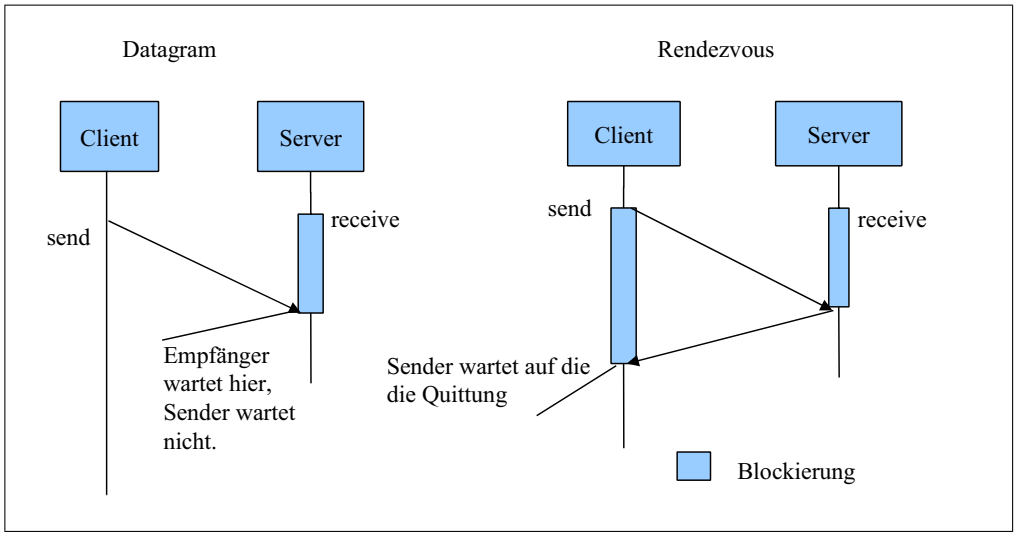

Abbildung 2.6.: Meldungsorientierte Kommunikation

2.3.2. Meldungs- und Auftragsorientierte Kommunikation

Während die meldungsorientierte Kommunikation eine Einwegnachricht ohne Antwort ist, stellt die auftragsorientierte Kommunikation einen Request/Response-Mechanismus dar, bei dem der Sender ein Ergebnis vom Empfänger erhält (entfernter Dienstaufruf). Bei Betrachtung der Tabelle 2.1 erschließen sich die verschiedenen Kombinationsmöglichkeiten zwischen synchroner/asynchroner und meldungs- /auftragsorientierter Kommunikation.
Zunächst wird die meldungsorientierte Kommunikation, wie in Abbildung 2.6 dargestellt, betrachtet. Hier kann die synchrone (Rendezvous) von der asynchronen Kommunikation unterschieden werden. Bei der synchronen Kommunikation sendet ein Client eine Nachricht an einen blockierten (wartenden) Server und der Vorteil bei Synchronisierung erfolgt dadurch, dass der Sender auf die Quittung, nicht aber auf ein Ergebnis des Servers, wartet. Bei der asynchronen Form der meldungsorientierten Kommunikation sendet der Client Datagramme, die nicht

bestätigt werden. Der Client wird dadurch auch nicht blockiert und kann sofort andere Dinge erledigen. Die Besonderheit ist hier, dass der Server eine *receive*-Primitive aufruft, um auf Datagramme zu warten, verarbeitet ankommende Datagramme und ruft dann wieder eine blockierende *receive* auf [MBW08].
Bei der auftragsorientierten Kommunikation werden ebenso synchrone und asynchrone Dienstaufrufe unterschieden. In Abbildung 2.7a ist die synchrone Variante dargestellt. Sie ist vergleichbar mit Rendezvous (synchronen meldungsorientierte Kommunikation). Im Unterschied zu Rendezvous wird bei der auftragsorientierten Kommunikation gleich ein Ergebnis geliefert. Der Client blockiert im blockierenden Fall, bis die Antwort ankommt. Die Bearbeitung des Auftrags kann sehr komplex sein. Es könnte sich z.B. um einen oder mehrere Datenbankzugriffe handeln, die abgewickelt werden müssen. In der asynchronen Variante (in Abbildung 2.7b) der auftragsorientierten Kommunikation kann der Client nach dem Versenden der Anfragen andere Aufgaben erledigen, muss sich allerdings irgendwann mit dem Server synchronisieren, was üblicherweise durch einen Aufruf eines *receive* Primitive erfolgt [MBW08].
Welche Kommunikationsform zu verwenden ist, hängt von der Anforderung der Anwendung ab. Wenn eine Client Anwendung erst nach dem Empfang eines Ergebnisses weiterarbeiten muss bzw. kann, ist eine blockierende Anfrage (synchron) zu empfehlen. Wenn der Client bis zum Empfang eines Ergebnisses etwas anderes tun kann bzw. muss oder gar keine Ergebnisse erwartet, ist eine asynchrone Variante zu nutzen. Es ist noch zu erwähnen, dass es die Voraussetzung ist, dass die hier angesprochenen Funktionsaufrufe bzw. Primitive (*send, receive*) durch eine Kommunikation-Middleware oder eine Transport-Zugriff-Schicht bereitgestellt werden müssen [Ben02].

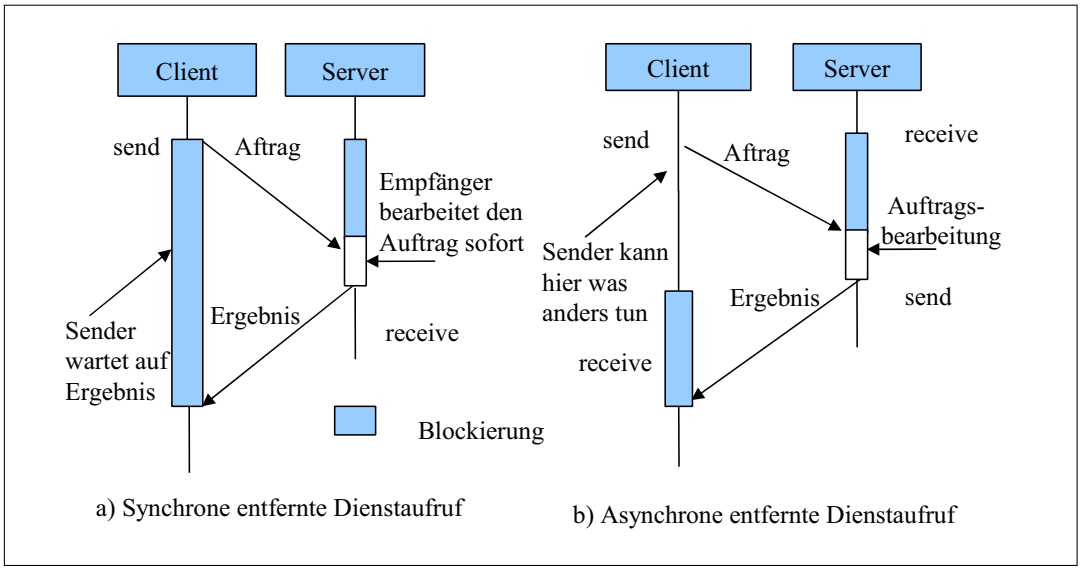

Abbildung 2.7.: Auftragsorientierte Kommunikation

2.4. Relevate Konzepte und Produkte

Wie bereits erwähnt, muss der TraceMaintainer zur Laufzeit mit verschiedenen SW-Anwendungen bzw. -Komponenten kommunizieren. Wenn diese Komponenten in der verteilten Umgebung angewendet werden sollen, muss ein effizientes und moderneres Kommunikationsverfahren entwickelt werden. Ein solches Verfahren sollte nicht nur den lokalen Datenaustausch zwischen den Komponenten, sondern auch die ortsunabhängige Kommunikation zwischen den Rechnern im Netzwerk unterstützen können. Bevor eine mögliche Lösung präsentiert wird, werden zunächst ein relevantes Konzept und ein dazu gehöriges Produkt vorgestellt.

2.4.1. Hub (Netzwerk)

Der Hub bezeichnet in der Telekommunikation Geräte, die Netzwerk-Knoten (physisch) sternförmig verbinden. Hubs werden im Bereich der ARCNet-Netzwerke eingesetzt, um eine Verteilung der Signale auf die einzelnen Anschlüsse vorzunehmen. Normalerweise wird die Bezeichnung Hub für Multiport-Repeater gebraucht [BH94]. Wie in Abbildung 2.8a gezeigt, werden sie verwendet, um Netz-Knoten oder auch weitere Hubs, z.B. durch ein Ethernet, miteinander zu verbinden.
Ein Hub besitzt nur Anschlüsse(Ports) mit gleicher Geschwindigkeit(mit gleichem MII, aber durchaus unterschiedlichem MDI). Ein Hub arbeitet, genau wie ein Repeater, auf Ebene 1 des ISO/OSI-Referenzmodells(Bitübertragungsschicht) und wird deswegen auch Multiport-Repeater oder Repeating-Hub genannt. Das Signal eines Netzteilnehmers wird in keinem Fall analysiert, sondern nur elektronisch aufgebessert(entrauscht und verstärkt) und im Gegensatz zum Switch, der sich zielgerichtet Ports des Empfängers sucht, an alle anderen Netzteilnehmer weitergeleitet. Hubs können in einem Ethernet nicht beliebig kaskadiert werden, um eine größere Netzausdehnung zu erreichen. Eine für jede Geschwindigkeit spezifische maximale Round-Trip-Delay-Time(RTDT) darf nicht überschritten werden. In der heutigen Netzwerktechnik kommen Hubs praktisch bei allen Netzwerktypen(Ethernet, Token-Ring usw.) zum Einsatz.

2.4.2. Hub and Spoke

Mit dem *Hub and Spoke* Konzept sind mehrere Partner (Unternehmen, Niederlassungen, Systeme, ...) durch Spokes (Speichen) über einen Hub (einer Nabe) verbunden. Das gegenteilige Konzept ist die Verbindung von „jedem mit jedem “ [Gmb87]. Unter dem *Hub and Spoke* versteht man im Transportwesen eine sternförmige Anordnung von Transportwegen, die den Transport wesentlich effizienter macht, indem ein Netzwerk von Strecken stark vereinfacht wird. Es findet vielfach in der gewerblichen Luftfahrt für den Personen- und Güterverkehr Anwendung. Auch im Technologie-Sektor wurde das Modell als gut bezeichnet. Das *Hub and Spoke* Verfahren hat sich in der kommerziellen Luftfahrt nach der Deregulierung des Luftfahrtwesens in den USA Ende der 70er Jahre in erster Linie aus wirtschaftlichen Gründen durchgesetzt und die bis dahin verbreiteten Punkt-zu-Punkt-Verbindungen stark zurückgedrängt [Sch99]. Die FedEx Unternehmen nutzten das Konzept, dass die Methode wirklich halten hat, revolutioniert die Fluggesellschaften [Exp07], [Qoo09].

Hub und Spoke Konzept werden in der Informatik auch verwendet, wenn mehrere Informationssysteme über eine zentrale Integrationsplattform verbunden sind, die Steuerung von Prozessen und der Informationsaustausch über diesen zentralen Hub erfolgt [Gmb87]. Wie in Abbildung 2.8b gezeigt, wird nach diesem Konzept der Weg von einem Endknoten A zu einem Endknoten B dabei nicht direkt, sondern über einen Zentralknoten Z, den Hub, geführt. Die Verbindungen der Endknoten A, B zum Knoten Z bezeichnet man hierbei als Spokes (dtsch.: Speiche). Der Transportweg zwischen zwei im Bereich verschiedener Hubs befindlichen Netzzugangsstellen führt auch dann über die beiden Hubs, wenn ein kürzerer, direkter Transportweg technisch möglich ist. Man unterscheidet Hub-and-Spoke-Systeme mit Einzelzuordnung (Single Allocation) und Mehrfachzuordnung (Multiple Allocation) [Sch99]. Bei Einfachzuordnung besitzt jede Quelle und jede Senke genau eine Verbindung zu genau einem Hub. Bei Mehrfachzuordnung können Quellen und Senken zu mehreren Hubs Verbindungen aufbauen.
Es gibt natürlich auch Nachteile mit sich, die Störung an der Nabe. Die allgemeine operative Effizienz ist auch begrenzt durch die Kapazität der Nabe [Qoo09].

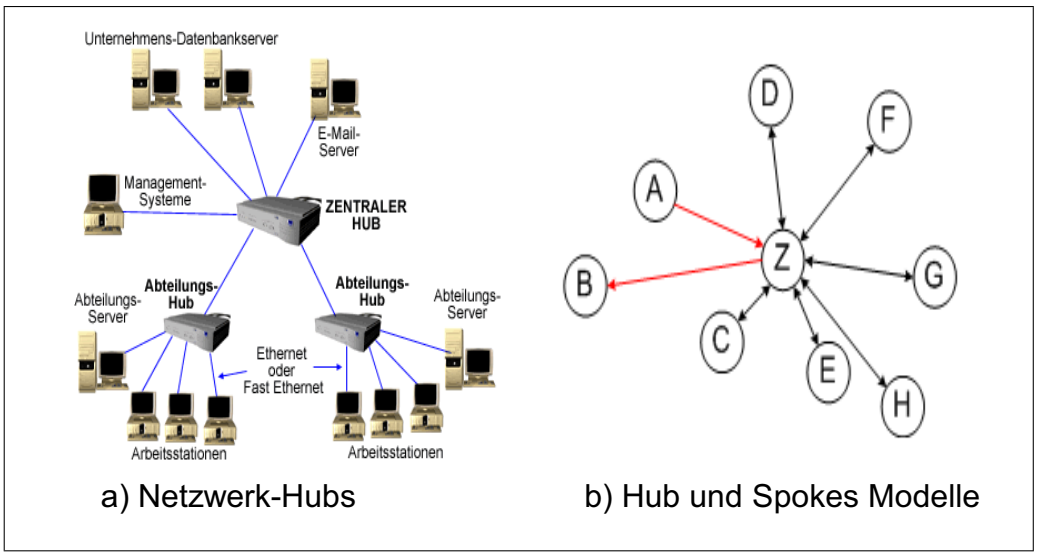

Abbildung 2.8.: Netzwerk-Hubs und Hub/Spoke Modell

2.4.3. Publish-Subscribe-Modell

Das Publish-Subscribe-Modell ist ein Entwurfsmuster aus dem Bereich der Softwareentwicklung und gehört zu der Kategorie der Verhaltensmuster (Behavioural Patterns). Es dient zur Weitergabe von Änderungen an einem Objekt an von diesem Objekt abhängige Strukturen [Zei04]. Dieses Entwurfsmuster ist auch unter dem frei übersetzten Namen „veröffentlichen und abonnieren “ bekannt. Das Publish-Subscribe-Modell ist ein asynchronesMany-to-Many Kommunikationsmodell für verteilte Systeme [Mue02]. Es gibt zwei verschiedene Typen von Clients in diesem Modell: Publisher (Anbieter) und Subscriber (Abonnenten).

- Publisher: sie sind diejenigen, die Ereignisse produzieren und publizieren

- Subscriber: sie erhalten die von Publishern veröffentlichten Ereignisse

Diese beiden bilden die Grundbausteine des Publish/Subscribe-Modells. Subscriber beschreiben die Ereignisse, welche sie erhalten wollen, anhand eines entsprechenden Abonnierungsmechanismus. Die von Publishern stammenden Ereignisse bzw. Nachrichten werden nach und nach an alle sich dafür interessierende Subscriber ausgeliefert. Publish-Subscribe-Modelle, wie in der Abbildung 2.9 dargestellt, bestehen aus drei Instanzen sowie Publisher, Subscriber und Broker (mit Message Channels). Es wird vorausgesetzt, dass die Events/ Messages, die hier verwendet werden, in entsprechend Messageformat kategorisiert sind (oder kategorisiert werden müssen). Eine sehr wichtige Eigenschaft ist, dass die Publisher und Subscriber nicht direkt miteinander verbunden sind. Ein Publisher braucht nicht alle Subscriber zu kennen und die Subscriber brauchen nicht die Identität der Anbieter zu kennen. Der Verlust dieser Verbindungen führt zu mehr Unabhängigkeit zwischen den Clients und zu einem skalierbaren Kommunikationsparadigma für groß skalierte Systeme [Mue02]. Ein PS-System besteht mindestens aus folgenden Instanzen [Zei04]:
Publisher sind für die Generierung von Nachrichten verantwortlich. Sie stellen die Ereignis-Anbieter dar. Die zu veröffentlichenden Ereignisse müssen von Subscribern abonniert werden.
Ereignis-Subscriber ind die Abonnenten. Sie müssen sich für bestimmte Events bzw. Messages abonnieren, so dass sie gewünschte Messages nach deren Veröffentlichung jederzeit empfangen können.

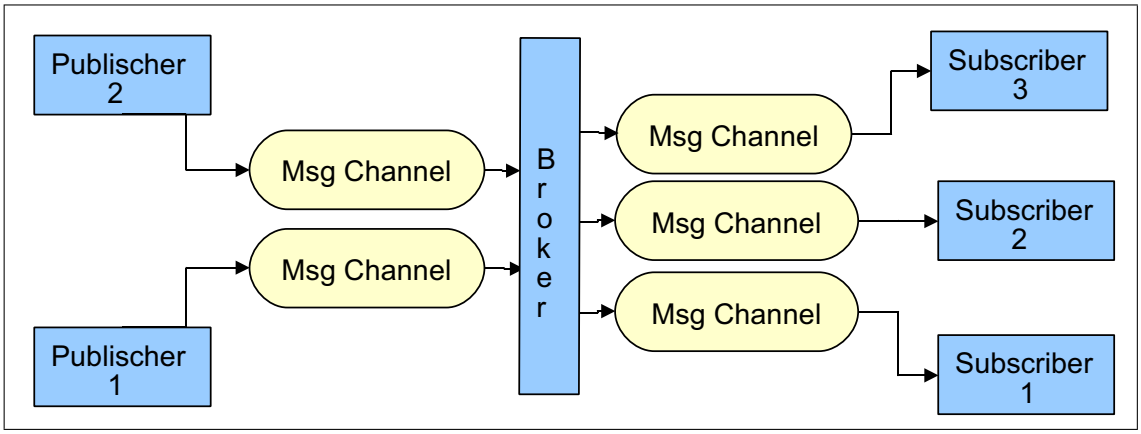

Abbildung 2.9.: Publish und Subscriber Modell

Ereignis-Broker ist der Vermittler und ein einfacher Message Router. Er leitet eine Message zu einem oder mehreren Subscribern weiter. Wie in Abbildung 2.9 dargestellt, verhält sich der Ereignis-Broker wie ein Vermittler: Er hört jede Message von einem abgehenden Message-Channel von einem Publisher, leitet die Messages anhand des Massageformats nach einem eingehenden Message-Channel von jedem Subscriber weiter. Durch ihre Anwendung der Ereignis-Broker sind Publisher und Subscriber abgekoppelt. Die Messsage-Channels sind während der Systemkonfiguration und Initialisierung angelegt. Damit die Nachrichten von Publishern an richtigen Subscriber verteilt wird, müssen einige Filterungsregel definiert werden. In [Zei04]

wurde 5 Arten von solchen Filterungs Modellen in Publish-Subscribe Systemen vorgestellt:
Channels: Das ist einfachste Form der Abonierung. In diesem Filterungsmodell muss der ein Publischer den Channelnamen auswählen, in den er seine Event veröffentlichen möchte. Zum Empfangen dieser Events muss der Subscriber einfach den Channel auswählen.
Themen-basiertes Publish/ Subscribe: Eine der frühesten Varianten des Publish-Subscribe-Modells. Die Anbieter veröffentlichten ihre Ereignisse unter bestimmten Themen. Nachdem die Abonnenten ihr Interesse an einem Thema bekundeten, erhielten sie die Ereignisse des betreffenden Themas.
Inhalte-basiertes Publish/ Subscribe: Die Struktur eines Ereignisses ist in einem Inhalte-basierten Publish-Subscribe-Modell ziemlich offen. Die Konsumenten definieren ihr Interesse nicht über Themen, sondern über Filter, die ihre Interessen beschreiben. Je komplizierter solch ein Filter definiert wird, um so schwieriger wird seine Evaluation. Es gibt hier einen Trade-off zwischen einem ausdrucksstarkem Filter und der Skalierbarkeit des Systems. Die Strukturen von Ereignissen können binäre Daten, Schlüssel/ Wert-Paare, halb strukturiertes XML oder sogar Klassen mit ausführbarem Code beinhalten.
Die zwei weiteren Modelle sind Typ-basiertes und Konzept-basietes PS-Modelle. Während das Typ-basierte PS-Modell gleiche Pfad-Ausdrücke nutzt, ist das Konzetp-basiertes PS-Modell eine Erweiterung von Inhalte-basiertes PS-Modell und ist besonders in den Umgebungen mit heterogenen datenquellen sehr nützlich.

3. Aktuelle Technologien für die Realisierung verteilter Anwendungen

Eine verteilte Anwendung bringt viele Vorteile mit sich. Der Trend hin zu verteilten Anwendungen nimmt deswegen zu und es gibt mittlerweile viele Technologien, um derartige Systeme zu realisieren [Ben02]. Als Beispiele hierfür sind COM/COM+, .Net Remoting, Webservices, Windows Kommunikation Foundation, Remote Methodenaufruf (RMI in Java), Enterprise JavaBeans,etc. zu nennen. Diese Technologien haben sich in den letzten Jahren sehr weit verbreitet und sind inzwischen Standard bei der Realisierung der verteilten Anwendungen geworden [Ill07]. Mit ihrer Hilfe ist es möglich, sehr schnell und effizient verteilte Anwendungen zu entwickeln, solange die Komplexität dieser Anwendungen in einem gewissen Rahmen bleibt. Ausschlag gebend für ihre Auswahl ist der Anwendungsfall [MBW08]. In diesem Kapitel wird ein Überblick zu ausgewählten Themen aktuelle Technologien für die Realisierung der Kommunikation der verteilten Anwendungen gegeben.

3.1. Kommunikationsmethoden bei der Netzwerkprogrammierung

In einer monolithischen Anwendung finden Prozedur- und Funktionsaufrufe lokal statt. Dazu wird der aufgerufenen Funktion eine Parameterliste übergeben, in deren Abhängigkeit dann das Ergebnis der Funktion berechnet wird. Der Rückgabewert der Funktion entspricht dann der Antwort, ist also das Ergebnis des Funktionsaufrufs [Ben02]. Ein großer Nachteil dieser Kommunikationsart ist die Tatsache, dass sie nicht über ein Netzwerk erfolgen konnte. Deshalb wurden verschiedene Möglichkeiten geschaffen, eine solche Verbindung zu ermöglichen. Um die nachrichtenorientierten Systeme bzw. Anwendungen als Teil von Middleware-Lösungen besser zu verstehen und einschätzen zu können, wird in diesem Abschnitt die Nachrichtenübergabe über Sockets und entfernten Prozeduraufruf vorgestellt.
Socket-Programmierung: Sie ist der erste Quasi-Standard. Zu Beginn der Verteilung von Programmen auf mehrere Rechnerknoten eines Netzwerks haben Sockets diese Möglichkeit angeboten, so dass eine Trennung von dienst-anbietenden(Server) und dienst-nutzenden Applikationen(Client) ermöglicht ist [KCH04], [Haa01]. Vom Konzept her ist ein Socket ein Kommunikationsendpunkt, an den eine Anwendung (Server oder Client) Daten schreiben kann, die über das zugrunde liegende Netzwerk versendet werden sollen. Außerdem können von diesem Endpunkt die eingehenden Daten gelesen werden können [TS03]. Bei der Kommunikation zwi-

schen Server und Clients über Sockets verfolgt eine Server Applikation auf einem bestimmten Rechnerknoten und die Netzwerkkommunikation erfolgt an einem bestimmten Port. Ein Client baut zu diesem Server unter Angabe von Netzwerkadresse und Portnummer eine Verbindung auf, über die er einen Bytestrom mit einer codierten Anfrage sendet. Für die Anfrage muss zuvor ein Protokoll und ein spezielles Datenformat verwendet werden. Diese Anfrage wird vom Server entgegen genommen und decodiert. Dann verarbeitet der Server sie und sendet eine entsprechende, ebenfalls codierte Antwort an den Client zurück [Haa01]. Dieses Verfahren bringt jedoch auch viele Nachteile mit sich.
Der größte Nachteil bei der Entwicklung der Socket Programmierung ist die Intransparanz der Netzwerkkommunikation, d.h. der Aufruf eines Server-Dienstes kann nicht, wie von der lokalen Programmierung her gewohnt, durch einen einfachen Prozeduraufruf geschehen, sondern erfolgt über einen Datenstrom, der geeignet codiert sein muss. Hierbei ist nicht nur die Überlegungen zu treffen, wie die zu übertragenden Daten serialisiert werden müssen, auch die Implementation der (De-) Codierung ist sehr fehleranfällig [KCH04].
Remote Procedure Call (RPC) : Ein bedeutender Schritt in Richtung Transparenz wurde durch RPC geschaffen. Das ist ein grundlegender Mechanismus für verteilte Systeme und in [BN84] wurde von *Birrel und Nelson* es wie folgt definiert:

> „RPC ist ein synchroner Mechanismus, der Kontrollfluss und Daten als Prozeduraufruf zwischen zwei Adressräumen über ein Netz transferiert. “

Somit wurde den Programmen erlaubt, Prozeduren aufzurufen, die sich nicht auf anderen Rechnern befinden. Wenn ein Prozess auf Rechner A eine Prozedur auf Rechner B aufruft, wird der aufrufende Prozess auf A unterbrochen, und die Ausführung der aufgerufenen Prozedur findet auf B statt. Während dieser Zeit ist keine Nachrichtenübergabe für den Programmierer sichtbar [TS03]. Das eigentliche Datenaustauschformat zwischen den Prozessen (Client und Server) ist hierbei systemweit standardisiert (SunRPC, DCE-RPC)und tritt es deswegen überhaupt nicht mehr bei der Softwareentwicklung in Erscheinung [KCH04]. Viel mehr werden die Dienste des Servers in einer Schnittstelle beschrieben. Diese Schnittstellenbeschreibung, die i.d.R, durch die Schnittstellenbeschreibungssprache IDL (Interface Definition Language) realisiert wird, kann im Client genutzt werden, um verteilte Procedure-Aufrufe an einen Server -ähnlich lokal ablaufenden-abzusetzen . Durch Werkzeuge werden Client und Server um lokale Stellvertreter des Interaktionspartners (stubs) ergänzt,die die technische Übermittlung der notwendigen Daten, wie deren Serialisierung der Netzwerkübertragung übernehmen [TS03]. Ein beim heutigen Stand der Technik wesentlich größerer Nachteile von RPC ist jedoch die Tatsache, dass RPC Verfahren nicht objektorientiert sind [KCH04].

3.2. Verteilte Objekte und Middlewareplattformen

In diesem Abschnitt wird das Thema von der Beschreibung der Grundlage zur Betrachtung verschiedener Paradigmen, die beim Aufbau verteilter System angewendet werden.
Als wesentlicher Nachteil von RPC beim heutigen Stand der Technik ist zu nennen, dass es sich um kein objektorientiertes Verfahren handelt. Dieser Nachteil wurde durch die Einführung objektorientierter Middleware Plattformen für verteilte Anwendungen wie CORBA, RMI, COM/DCOM oder auch .Net beseitigt [KCH04], [MNW03]. Bei der Entwicklung verteilter

Objekte und Anwendungen mit diesen Plattformen werden - ebenfalls wie bereits gewohnt, die Schnittstellen wie bei RPC beschrieben, allerdings verläuft die gesamte Arbeit nun objektorientiert. Im Rahmen dieser Diplomarbeit für die Realisierung des RemotingHub wurden Sockets, RPCs und alle o.g. Technologien untersucht. Im Ergebnis dieser Untersuchung wurde entschieden, .Net Technologie einzusetzen. Der Grund dafür ist, dass der TraceMaintainer mit Hilfe der .Net Technologie entwickelt wurde. Allein aus dem Kompatibilitätsgrund heraus macht es keinen Sinn, RemotingHub mit einer von .Net verschiedenen Middlewareplattformen zu realisieren. Für die nachfolgend genannte Auswahl an Argumenten waren folgende Punkte Ausschlag gebend:

1. .Net bietet außerdem gleiche oder sogar bessere Technologische Unterstützung bei der Realisierung einer verteilten Anwendung.
2. Dazu kommt noch, dass die RMI ein Java-spezifisches Kommunikationsprotokoll besitzt. Es handelt sich dabei nicht um eine Interface-Beschreibungssprache, die in unterschiedlichen Zielsprachen überführt werden kann, sondern um Java-Interfaces, aus denen auch nur Java-kompatible Stubs generiert werden. Das bedeutet, dass Java RMI nur mit der Programmiersprache Java nutzbar ist [KCH04].
3. Enterprise JavaBeans (EJBs) sind eingebettete, verteilte Java-Komponenten für J2EE, die auf einem Server ablaufen und Clients den Zugriff auf Ihre Funktionalität bieten. Der Server bietet den Komponenten je nach Typ die Dienste sowie Transaktionsmanagement und automatisierte Persistenzhaltung, so dass sie die Entwicklung komplexer mehrschichtiger verteilter Softwaresysteme mittels Java vereinfachen [Jav], [EF03], [KCH04]. Deswegen sind EJBs für die Realisierung des RemotingHub überdimensioniert.
4. D/COM+ ist eine von Microsoft entwickelte Plattformtechnik, um unter dem Betriebssystem Windows Interprozesskommunikation und dynamische Objekterzeugung zu ermöglichen. COM+ ist die moderne Konzeption für die Entwicklung von komponentenbasierten, verteilten Systemen. Allerdings gibt es viele Probleme. Deswegen ist .NET Framework ein guter Ansatz, um die Probleme zu lösen, die bei COM+ auftreten [MSDa].
5. CORBA ist relative veraltete Technologie im Vergleich zu .Net. Es war sogar Vorgänger von D/COM.

.Net Technologie ist auf jeden Fall in Bezug auf die Interoperabilität von Programmiersprachen und im Hinblick auf die Interoperabilität verschiedener Plattformen zukunftsweisend [MSDc]. Auch das Deployment von Anwendungen ist in .Net deutlich einfacher. Wegen der Vielseitigkeit der .Net-Plattform existieren mehrere unterschiedliche Möglichkeiten, verteilte Systeme zu realisieren. Repräsentative Standardtechniken sind ASP.Net, .Net Enterprise Services, Webdienste, .Net Remoting, WCF. Nach dem Untersuchungsergebnis werden ASP.Net, .Net Enterprise Services für die Realisierung von RemotingHub als ungeeignet aussortiert. Deswegen werden nur die letzten drei Techniken vorstellen.

3.3. .Net Remoting

Das .Net Remoting ist der Nachfolger von COM/DCOM und bietet eine vergleichbare Fähigkeiten zum Aufbau verteilter Systeme [MSDd]. Mit Hilfe des Remoting Frameworks können verteilte Objekte über Prozess- und Rechnergrenzen hinweg verwendet werden. Die Interaktion zwischen verteiltem Objekt und dem Client läuft über einen Kanal. Die beteiligten Elemente sind der Server, der eine Instanz eines verteilten Objektes hält, und der Client, der mit Hilfe des Proxy mit dem verteilten Objekt über den Kanal kommuniziert. Wie in Abbildung 3.1 zu sehen ist, handelt es sich bei der clientseitigen Instanz der remotefähigen Klasse um ein Abbild des eigentlichen Objekts, einen sogenannten Proxy [KCH04], [Ram02], [MNW03]. Der Proxy hat dabei das gleiche Erscheinungsbild wie das Objekt auf dem Server, dient aber lediglich als Verweis auf das serverseitige Objekt. Der Zugriff auf die Funktionen des Proxy-Objekts erfolgt für den Programmierer vollkommen transparent. Für den Kommunikationskanal zwischen dem Proxy-Objekt auf dem Client und dem Remoteobjekt auf der Serverseite existieren innerhalb des .NET-Frameworks zwei Varianten. Das ist zum einen der TCP-Kanal auf der Basis des TCP/IP-Protokolls und zum anderen der http-Kanal, der auf dem http-Protokoll beruht. Die Wahl des jeweiligen Kanals ist dem Programmierer freigestellt. Neben dem binären Kommunikationsprotokoll und dem SOAP-Protokoll kann auch das ORPC von COM/DCOM verwendet werden. Des weiteren ist es möglich, eigene Kommunikationsprotokolle zu definieren [KCH04].

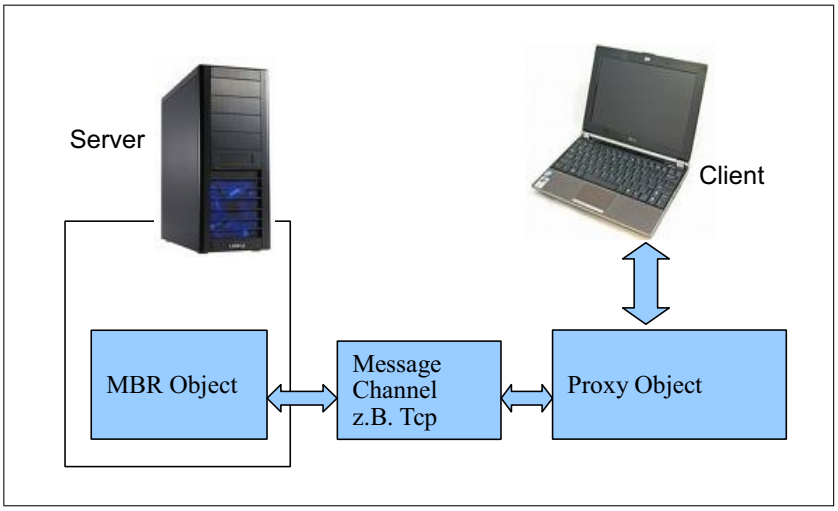

Abbildung 3.1.: Das Anwendungsprinzip des .Net Remoting

Vorteil von .NET Remoting ist die Integration in das .NET Framework und die damit verbundene gute Anbindung an Programme, die auf diesem aufbauen [Ram02], [MSDd]. Weiterhin stellt die Verbindung auf Basis des TCP-Protokolls eine schlanke, schnelle Übertragung von Nachrichten und Parametern innerhalb der verteilten Anwendung sicher. Die Verwendung eines TCP-Kanals unter Angabe eines Ports kann aber auch zu einem Nachteil von .NET Remoting werden: Die Verbindung kann an Rechnergrenzen durch Firewalls blockiert werden. Eine

Verbindung ist dann nicht oder nur unter hohem Administrationsaufwand möglich. Ein weiterer Nachteil ist die Tatsache, dass .NET Remoting nur innerhalb des .NET Framework lauffähig ist. Eine Plattformunabhängigkeit ist damit nicht gewährleistet. Client und Server müssen das .NET Framework implementieren. Eine nähere Beschreibung der Funktionsweise von .NET Remoting erfolgt an einem konkreten Beispiel im Kapitel 7 in Zusammenhang mit RemotingHub.
NET Remoting führt standardmäßig keine Authentifizierung oder Verschlüsselung durch. Daher empfiehlt es sich, vor der Remoteinteraktion mit Clients und Servern alle erforderlichen Schritte zu unternehmen, um die Identität der Clients oder Server zu überprüfen [KCH04]. Deswegen sollten unbedingt Endpunkte authentifizieren und verschlüsseln werden, indem die jeweiligen Remotingtypen in Internetinformationsdiensten (IIS) gehostet werden oder ein angepasstes Channel-Empfänger-Paar erstellt wird, das diese Aufgabe übernimmt [MSDN]. .NET Remoting wird von vielen Verfechtern der losen serviceorientierten Kopplung wegen seiner engen Kopplung kritisiert. In der in .Net 3.0 enthaltenen Kommunikationsinfrastruktur WCF (Windows Communication Foundation) wird das .NET Remoting Modell nur noch in wenigen Teilen vertreten sein. Eine Vorstellung der WCF erfolgt im Abschnitt 3.5 dieser Arbeit.

3.4. Web Services

Webdienste stellen eine Technologie für interoperabel verteilte Anwendungen dar, die an SOAP und XML anknüpfen und in den letzten Jahren enorm an Bedeutung gewonnen haben [Ill07]. Oft werden sie auch noch XML-Webdienste/ Webdienste oder Webservices genannt. Webdienste sind ein Service, werden auf einem Rechner im Netzwerk gehostet wird und sind über einen URI erreichbar. Sie besitzen keine Benutzeroberfläche, sondern stellen ihre Dienste in Form von Webmethoden im Internet bereit. Die Webmethoden werden vom Client der Anwendung mittels http-Anfragen aufgerufen. Die Antwort des Servers ist nicht wie bei ASP eine HTML-Seite, sondern eine Nachricht in Form von XML-Daten [MSDe] . Auf diese Weise können einzelne Dienste auch zu komplexen Diensten komponiert werden. Da Webdienste keine Oberfläche besitzen, können für den Zugriff auf die Webmethoden erstellten Clients neben Windows-Programmen, Webseiten oder Konsolenprogrammen auch weitere beliebige Anwendungen eingesetzt werden, die XML-Daten auswerten können. Webdienste bieten also eine ideale Möglichkeit, um plattformunabhängig verteilte Anwendungen zu schreiben [EF03].
Obwohl Web Services auf bekannten „Internet-Technologien “ basieren, unterscheidet sich die dahinter stehende Architektur grundlegend von der klassischen Client/Server Architektur des Internets [Ill07]. Dort greifen die Clients auf bestimmte Daten bzw. auf die Funktionalität eines Servers zu und stellen diese im Browser dar. Bei Web Services hingegen findet eine Maschine-zu-Maschine Kommunikation statt. Hier werden Daten bzw. Funktionalität zwischen Anwendungen ausgetauscht. Web Services lassen sich durch die so genannte service-orientierte Architektur (SOA) [EF03], [Ill07] beschreiben. Diese Architektur in Abbildung 3.2 basiert auf folgendem Prinzip: Ein Service-Anbieter stellt einen Service zur Verfügung. Um diesen bekannt zu machen veröffentlicht er ihn in einem Service-Verzeichnis. Der Service-Konsument kann nun in diesem Verzeichnis nach dem Service suchen (Schritt 2), den er beanspruchen will. Dort wird ein passender Dienst gefunden und dessen WSDL-Beschreibung sowie weitere Informationen zurückgeliefert (Schritt 3). Beim Aufruf (Schritt 4) einer Operation eines Webservices

sowie für die Übermittlung des Ergebnisses wird jeweils ein SOAP Message verschickt. Der eigentliche Kommunikationsakt besteht nun darin, dass ein Client eine Methode in herkömmlicher Art aufruft und diese von einem Proxy entgegengenommen wird, welcher zuvor aus der WSDL Beschreibung des genutzten Dienstes erzeugt wurde und in die SOAP Anfrage konvertiert. Sie wird an den Server weiter geschickt und dort vom SOAP-Dispatcher wieder durch einen herkömmlichen Methodenaufruf an die Dienstkomponente umgewandelt. Der Rückweg der Antwort läuft analog zu Schritt 4 in der Abbildung 3.2 [KCH04].

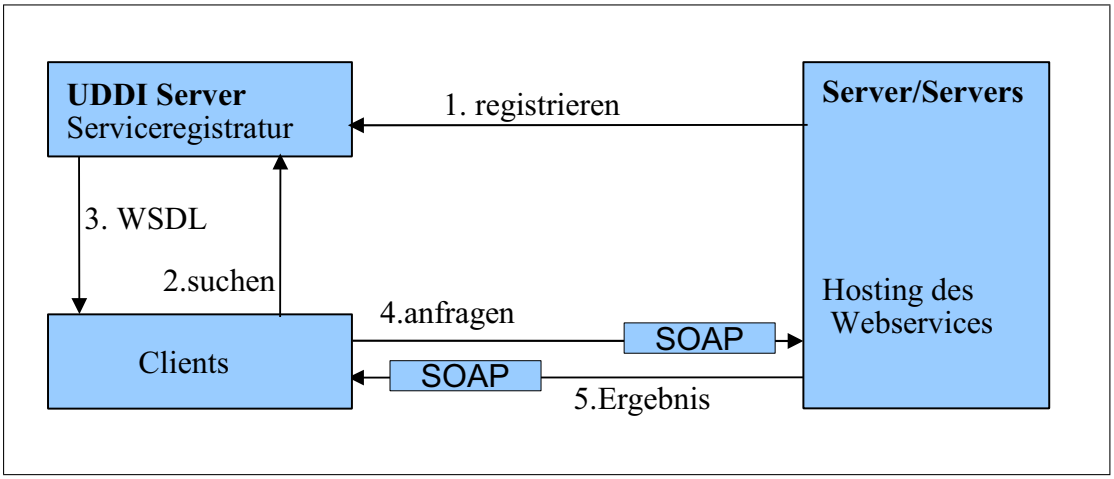

Abbildung 3.2.: Allgemeiner Ablauf des Webdiensts

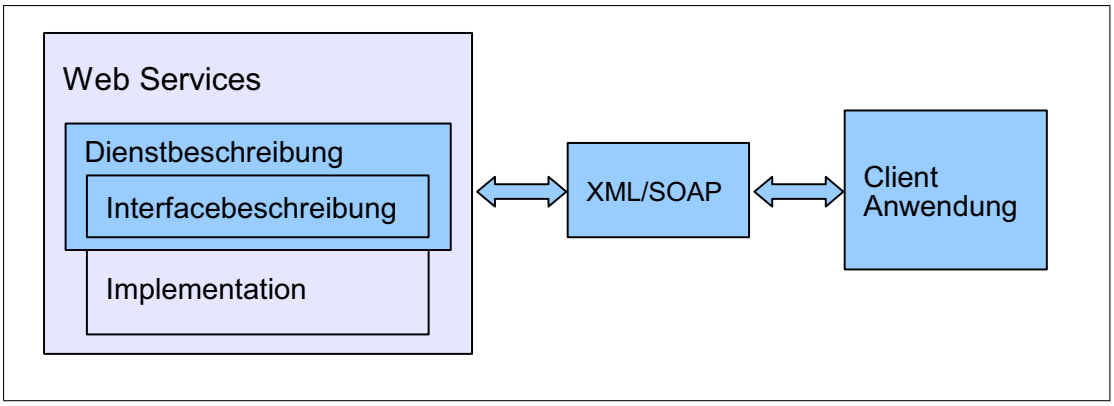

Abbildung 3.3.: Webdienste und Ihre Anwendung

Vereinfacht ausgedrückt stellt ein Anbieter über einen WebService einen Dienst bereit, der von externen Anwendungen angesprochen werden kann. Die standardisierten Protokolle und Formate wie XML, WSDL sowie Http sind die Basis von den Webdiensten. Sie stellen eine offene

Plattformen dar. Server können mit Hilfe von WSDL Beschreibungen als Dienste bzw. Webservices anzieten und können sie von den Clients genutzt werden, ohne Details über den Server zu kennen [KB07]. Weiterhin ist es möglich, ein komplexes verteiltes System hinter einem Webdienst zu verstecken und somit dessen Fähigkeiten beliebigen Clients zugänglich zu machen. Der Webservice kann somit auf eine bereits extrahierende n-schichtige Architektur aufsetzen und diese erweitern. Diese Gründe machen Webdienste sehr interessant, da sie sehr elegant viele betriebswirtschaftliche Anforderungen an verteilte Systeme, wie z.B. Investitionsschutz, Zahlungsverkehr oder Erreichbarkeit eines größeren Kundenkreises, realisieren können [KCH04]. In Abbildung 3.3 ist der allgemeinen Aufbau eines webservice-basierten Systems mit seinen relevanten Teilen zu sehen. Links in der Abbildung wird die Architektur eines Webdienstes erkenntlich, in der Mitte die Kommunikation und im rechten Teil der Nutzer. Kern eines Webservices ist naturgemäß seine Implementation. Diese erfolgt in einer Programmiersprache, die der späteren Hostumgebung entspricht. Außer der Implementation eines Webdiensts muss noch dessen Dienst beschrieben werden. Die Interface-Beschreibung ist der obligatorische Teil der Dienstbeschreibung. Sie erfolgt in WSDL und bietet syntaktische Informationen zur Nutzung eines Webdienstes. Eine Beschreibung in WSDL beinhaltet Message Abschnitte, Porttyp Abschnitte und die Ports selbst. Die Dienstbeschreibung beinhaltet die Bindung der Ports an den Dienst. Ein weiterer sehr wichtiger Aspekt: In diesem Abschnitt wird außerdem die Internetadresse festgelegt, unter deren Angabe der Webdienst erreichbar ist [EF03], [KCH04]. Der letzte Aspekt der Webservices-Technologie an dieser Stelle ist die Kommunikation zwischen einer dienstnutzenden Anwendung und dem Webservice selbst.

3.5. Windows Communication Foundation

Wer bislang die Aufgabe lösen mochte, seine Windowsanwendung mit anderen Prozessen kommunizieren zu lassen, hatte die Wahl zwischen einer Vielzahl verschiedener Möglichkeiten: Enterprise Services, System-Messaging, WSE oder .NET Remoting. Je nachdem, ob die Kommunikation zwischen Applikationen auf der gleichen Maschine, im LAN oder über das Internet stattfand, ob synchron oder asynchron, uni- oder bidirektional, verschlüsselt, als Binär- oder XML-Datenstrom, ob auf beiden Seiten die gleiche oder verschiedene Plattformen zum Einsatz kamen. Mit der Fertigstellung der Windows Communication Foundation (WCF, vorher Indigo) als Teil des .NET Framework 3.0 wird dies anders [MSD05]. Die WCF fasst alle bestehenden Kommunikationsansätze von Microsoft unter dem Dach eines einheitlichen APIs zusammen. WCF ist mehr als ein gemeinsames API über bestehenden Kommunikationsstacks. Es kombiniert die Vorteile der verschiedenen Technologien. WCF ist in dem Sinne noch ein weiterer, moderner Satz von .NET-Technologien zum Erstellen und Ausführen vernetzter Systeme [Prö07]. WCF erleichtert jedoch dem Entwickler nicht nur die Arbeit, es zwingt ihn auch dazu, expliziter als bisher zu beschreiben, was er will. Wie Abbildung 3.4 zeigt, lässt sich das WCF Konzept des Endpunktes durch eine Trennung in Address, Binding und Contract abstrahiert werden. Das ABC-Prinzip beschreibt, wie die Dienstleistungen erreichbar und nutzbar sind. Ein Kommunikationskanal wird zwischen je einem serverseitigen und clientseitigen Endpunkt aufgebaut [KB07]. Es beschreibt vollständig das Verhalten einer WCF-Applikation gegenüber der Außenwelt.

- Die **A**ddress (Adresse) ist ein URI, der den Ort des Dienstes beschreibt und somit seine Erreichbarkeit für die Dienstkonsumenten kennzeichnet.
- Das **B**inding (Anbindung) beschreibt die Art der Kommunikation, worunter unter anderem die Merkmale der Kodierung und des Protokolls fallen. Zum Binding gehören Parameter wie Protokolle(HTTP, TCP, UDP und Windows-eigene Protokolle) und Kodierung (binäre, SOAP, eigenes Kodierungsformat) sowie Sicherheitsaspekte (Verschlüsselung, Authentifizierung). Das .NET Framework stellt vorgefertigte Bindungen für häufige Anwendungsfälle zur Verfügung, die noch konfiguriert werden können. Wie schon erwähnt, besteht die Möglichkeit, eigene Bindings zu entwickeln.
- Der **C**ontract (Vertrag) stellt die Dienstdefinition, insbesondere die zur Verfügung gestellten Methoden beinhalten, dar. Verträge werden zur Entwicklungszeit als Interfaces (Schnittstellen) in einer beliebigen .NET-Sprache verfasst und zur Laufzeit durch die WCF in ein Kommunikationsprotokoll z.B. SOAP umgesetzt. Die Verwendung dieses Standards ist maßgeblich für einen plattformunabhängigen Dienstzugriff.

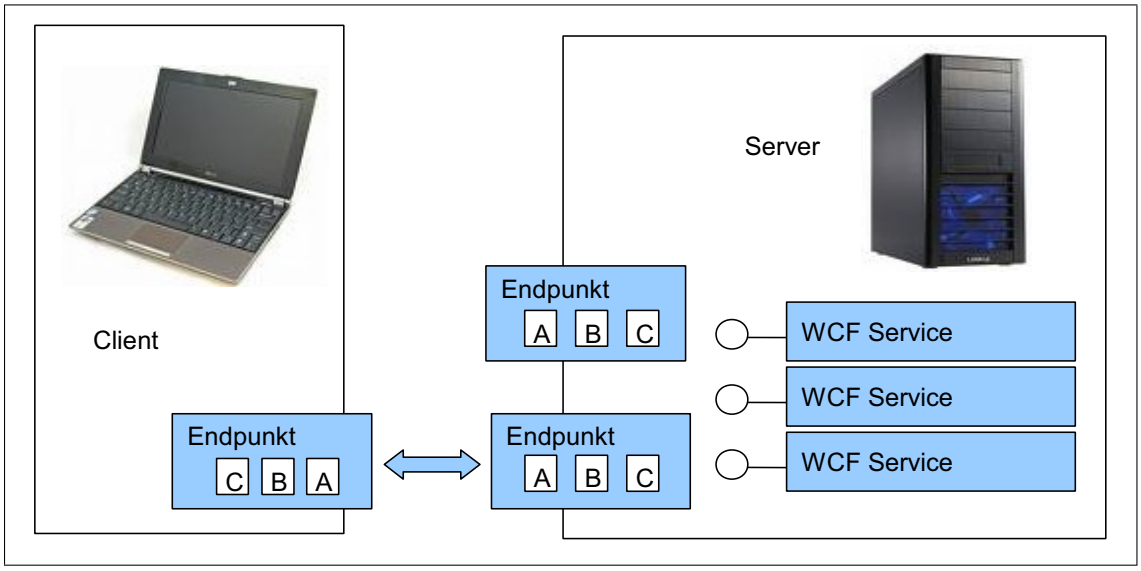

Abbildung 3.4.: Das ABC-Konzept der WCF

Adressen und Bindings sind dabei unabhängig von einem Service. Ein Service exportiert eine oder mehrere Schnittstellen als Verträge [KB07]. Jedem Vertrag können eine oder mehrere Endpunkte zugewiesen werden. Damit können sie für denselben Service verschiedene Interaktionsmöglichkeiten anbieten. Mit WCF lassen sich serviceorientierte Architekturen (SOA) implementieren. Der Vorteil gegenüber bestehenden Technologien ist darin zu sehen, dass die eigentliche Schnittstelle vom Nachrichtenformat und dem Transportweg unabhängig ist. Es ist einfach durch Konfiguration bestimmbar, ob die Kommunikation über HTTP, TCP, MSMQ, Named Pipes etc. stattfinden soll und ob der Nachrichtenaustausch über SOAP/XML, Binärserialisierung etc. erfolgen soll [MSD05].

3.6. Gegenüberstellungen

3.6.1. .Net Remoting und Webdienste

Webdienste sind sehr gut geeignet für serviceorientierte Anwendungen. Als Empfehlung gilt die Verwendung von Webservices [KCH04] , sollen Dienste oder sonstige Funktionalität im Internet veröffentlicht werden. Auch wenn geplant ist, dem System weitere Bestandteile hinzuzufügen oder es in andere Systeme zu integrieren, sind Webservices dafür richtige Wahl. Sie bietet die Möglichkeit, einfach und unkompliziert interoperable Dienste zu erstellen, die ohne größere Schwierigkeiten in heterogene Systeme integriert werden können.
.NET Remoting wird verwendet, wenn in einer Intranetanwendung ausschließlich eine leichte, schnelle CLR-basierte RPC-Kommunikation benötigt wird. Auch wenn Typsicherheit, statusbehaftete Objekte, etc. benötigt werden, sollte .NET Remoting den Web Services vorgezogen werden. Da die Objekte als Binärdaten übertragen werden, ergibt sich gegenüber Web Services meistens ein Geschwindigkeitsvorteil. Mit einer hybriden Lösung kann ein besserer Effekt erzielt werden.

3.6.2. .Net Remoting und WCF

Die Paradigmen von .NET Remoting und der WCF unterscheiden sich in einigen wesentlichen Punkten. Während .NET Remoting versucht, die Interprozesskommunikation für den Entwickler so einfach und transparent wie möglich zu machen, verfolgt die WCF eine andere Strategie. Sie verlangt vom Entwickler, über detaillierte Kontrakte (Contracts) explizit anzugeben, was er wie sichtbar und serialisierbar machen möchte [KCH04].
Dieser Paradigmenwechsel hat einen Grund. Unter .NET Remoting ist es sehr leicht, Objekte vom Server auf einen Client zu remoten und dort so zu verwenden wie eine lokale Objektinstanz. Der Unterschied zwischen lokalen und Remote-Objekten ist für den Entwickler nicht mehr zwangsläufig transparent. Deswegen gibt es immer wieder sehr ineffiziente Softwaredesigns, in denen z.B. Tausende von Remoting Calls pro Sekunde abgesetzt werden, nur um die Eigenschaften aus Remote-Objekten abzufragen. Da Problem von .NET Remoting liegt nicht darin, dass es nicht effizient genug funktioniert, sondern dass es den Entwicklern eine Transparenz und Leichtigkeit im Aufruf von Remote-Objekten vortäuscht (vorgaukelt), die es so nicht gibt. .NET Remoting möchte die Schwierigkeiten bei der Kommunikation über Anwendungs-, Rechner- und Netzwerkgrenzen vor dem Entwickler verbergen [Prö07]. Bestehende Applikationen, die .NET Remoting verwenden, laufen natürlich auch unter .NET 3.0 und werden weiterhin unterstützt. Gerade in begrenzten, lokalen Kommunikationsszenarien kann Remoting weiterhin das Mittel der Wahl sein. Manche Dinge wie die Übergabe von Objekten sind via Remoting einfach zu realisieren [Prö07]. Das Remote Objekt in WCF ist nicht gleich das Remote Objekt in .Net Remoting. d.h. WCF Remoteobjekte können nicht alles unterstützen, was die Remote Objekte in .Net Remoting tun [KB07].

- SOAP-Interoperabilität
- Server Activated Objects (SAO)

- Metadaten Import und Export
- Remoting Interceptionsmodell

Die Art von Konstruktionen, in denen über den Aufruf eines Remote Interface eine auf dem Server erzeugte Objektinstanz zum Client übertragen und von dort wie ein lokales Objekt verwendet werden kann, sind zwar auch unter WCF möglich, aber deutlich schwerer zu realisieren. So ist es keineswegs immer notwendig oder sinnvoll, eine auf .NET Remoting beruhende Anwendung zu WCF zu migrieren. Wer aber an die Grenzen von .NET Remoting stößt und vor der Aufgabe steht, aus seiner Remoting-Applikation eine WCF-Applikation zu machen, dem ist die Bücher [KB07], [Ram02] empfohlen, die die wesentlichen Schritte der Migration beschreiben.

4. Das Link-Nachführungssystem – TraceMaintainer

Heutige Softwareprojekte sind extrem komplex. Da man bei komplexeren Softwaresystemen mit hoher Anzahl von Komponenten sehr schnell den Überblick auf das gesamte System verlieren kann, ist jede Art der Änderung bei der Entwicklung sehr schwierig. Im Gegensatz dazu ist es jedoch bei solch großen Softwareprojekten meist nicht vermeidbar, immer wieder Änderungen vorzunehmen. In der Praxis hat man deswegen immer großen Bedarf an Traceability Link Beziehungen an Anforderungen, Analysis, Design Modelle von Software Modellierungstools. Daher wurden in den vergangenen Jahren verschiedene Methoden entwickelt, die Übersichtlichkeit durch Rückverfolgung über mehrere Phasen hinweg zu ermöglichen. Obwohl sie den Softwareentwicklungsteams bei der Entwicklungsarbeit helfen und den Software-Entwicklungsaufwand reduzieren, können sie dennoch keine überschaubare Entwicklungsarbeit zwischen Softwareanalyse und Design garantieren [Kus07]. Es mangelt sozusagen immer noch an der technischen Unterstützung solcher Tools. Aus diesem Grund entstand an der TU Ilmenau ein Projekt zur automatischen Nachführung von Traceability-Relationen (Traceability Links). Das Tool wurde von dem Forschungsteam TraceMaintainer (TM) genannt. In den folgenden Abschnitten soll das Projekt kurz vorgestellt werden. Weiterführende Informationen zum Projekt und dessen Stand lassen sich unter [Ilm06] einsehen.

4.1. Das System und dessen Struktur

Der TraceMaintainer ist ein SW-Tool, das dafür entwickelt wurde, sowohl die Traceability Links zwischen Analyse- und Designobjekten im Software-Entwicklungsprozess möglichst automatisch nach zuführen als auch die Modellierung eines ganzen Prozesses zu realisieren [MGP09a], [MGP09b]. Die Aktualisierungen der Traceability-Beziehungen erfolgt anhand von vorher definierten Regeln. Jede dieser Regeln wird zu den Entwicklungsaktivitäten auf einem Modellelement angewendet. Dazu ist nur sehr wenig manuelle Arbeit oder Interaktion mit dem Entwickler nötig. Dieses Tool füllt somit eine Lücke der Software Entwicklungstools (die mangelnde technische Unterstützung an Traceability-Link-Beziehungen bei der Softwareanalyse und ihr Design) und stellt eine softwaretechnische Realisierung der automatischen bzw. teil-automatischen Nachführung von Relationsänderungen zwischen Analyse- und Designobjekten dar.
Auf Abbildung 4.1 ist zu sehen, wie der TraceMaintainer aufgebaut ist und mit welchen Softwareentwicklungstools er zusammenarbeitet. Der TraceMaintainer besteht aus einigen Komponenten und ist so eingerichtet, dass er möglichst wenig von einem bestimmten Softwareentwicklungstool abhängig ist [MGP09a], [MGP09b]. Die zentrale Komponente RuleEngine (RE), dient dazu, die Aktivitäten zu erkennen. Es bietet eine Schnittstelle zum Empfangen neu-

er Änderungen im Modell und verlangt eine Schnittstelle zur Anfrage und Aktualisierung von Traceability-Beziehungen. Die Schnittstelle für Change-Events (Veränderungsaktivitäten) im Modell wird toolspezifisch vom Eventgenerator (EG) verwendet. Der EG erkennt dabei die Änderungen zu den Modellelementen, erzeugt die Events über diese Änderungen und schickt sie dann an RuleEngine. Die angeforderte Schnittstelle für Anfrage und Update muss auch von einem toolspezifischen Adapter implementiert werden. Mit dem Adapter kann auf die gespeicherten Traceability-Beziehungen auf dem Modell oder auf einem externen Tool sowie EXTESSY ToolNet zugegriffen werden. Darüber hinaus gibt es noch einen im XML-Format gespeicherten Regelkatalog, der von RuleEngine gelesen wird. Während der Regelkatalog (Rule Catalog) alle vordefinierten Regeln enthält, ist der Regeleditor eine eigenständige Anwendung, sodass er bei den Aufgaben wie Regelerzeugung, Editieren und Validierung hilft [MGP09a]. In den folgenden Abschnitten werden die Hauptkomponenten RE, EG und TS weiter vorgestellt.

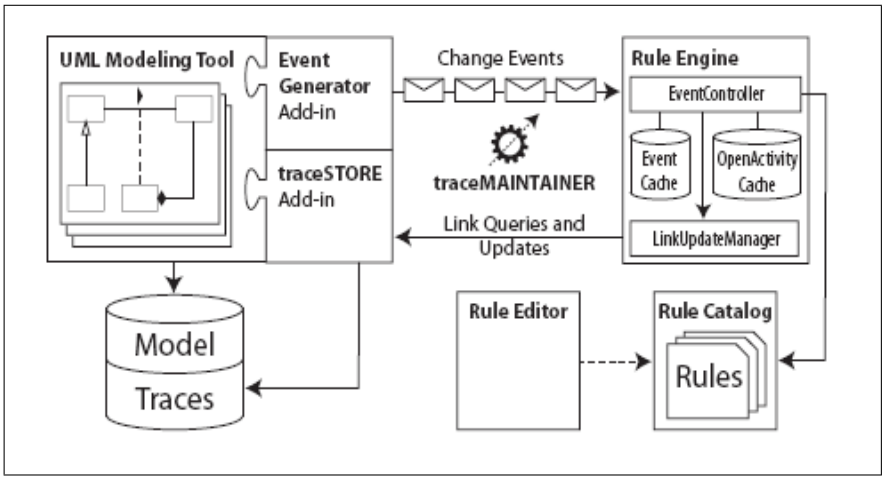

Abbildung 4.1.: Eine Überblick über die TraceMaintainer-Komponenten [MGP09b]

4.2. RuleEngine

Wie in Abbildung 4.1 gezeigt ist, beinhaltet RuleEngine drei Komponenten sowie EventCache, OpenEventActivity und LinkUpdateManager. Anhand eines klar definierten Regelkataloges erfolgen in RuleEngine eine Bewertung der Änderungen in der Modellierung und die sich daraus ergebene Nachführung der Traceability-Relationen. Aber es gibt Situationen, in denen der RuleEngine nicht allein über die Änderungen der Link-Beziehungen entscheiden kann [MGP09a], [MGP09a]. Deswegen hat RuleEngine eine einfache Benutzeroberfläche, die für normale Benutzer gedacht ist und in solchen bestimmten Situationen automatisch aufgeht, wo die Entwicklungsaktivitäten erkannt werden, aber nicht genügend Informationen vorhanden sind, um ein Traceability-Link-Update voll automatisiert durch zuziehen.
Wie Abbildung 4.2 zeigt, bietet dieser Dialog detaillierte Informationen über die erkannten

Entwicklungsaktivitäten und nötige Updates. Dieser Dialog zeigt zwei Arten von Listboxes sowie eingehenden (incoming) und ausgehenden (outgoing) Traceability-Link-Beziehungen, die getrennt angezeigt werden. Jeder dieser Zeilen zeigt eine vorhandene oder potenzielle neue Traceability-Link-Beziehung. Der User kann sich entscheiden, ob er die existierenden Links auf dem Source Element behalten (stay) oder löschen möchte. Es ist standardmäßig auf Behalten eingestellt. Für die geänderten Elements kann der User entscheiden, ob er die Link-Beziehungen neu erzeugen oder verwerfen möchte.

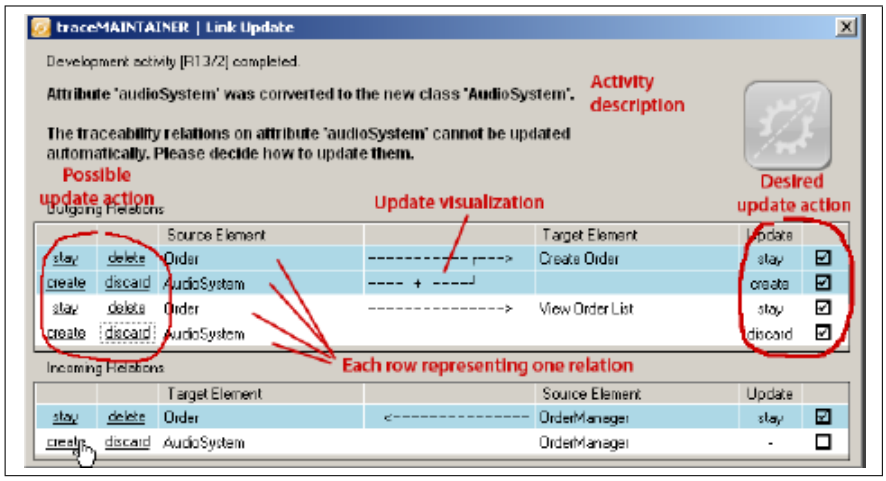

Abbildung 4.2.: GUI für die Nutzerinteraktion im TM [MGP09b]

4.3. Eventgenerator

Durch die Einbindung eines PlugIns in ein Modellierungstool hat man ermöglicht, die Events für RE zu erzeugen. Dieses PlugIn wurde vom Entwicklungsteam als Eventgenerator (EG) bezeichnet und ist eine spezifische Schnittstelle für das Modellierungstool und hat die Aufgabe, die Verbindung zwischen RuleEngine und dem Modellierungstool aufzubauen. Zur Zeit wurde er als PlugIn von Enterprise Architect entwickelt [MGP09a]. Um Change Events (Veränderungsaktivitäten) an die RuleEngine weiterzuleiten, beobachtet der EG alle Änderungen und die Arbeitsschritte an dem Modell, das gerade auf diesem EA bearbeitet wird. Der Ausgangspunkt zur automatischen Nachführung der Traceability-Link-Beziehungen ist, jedem dieser Änderungen bzw. Arbeitsschritte nun ein spezielles Event zuzuordnen und der RuleEngine zu übergeben [MGP09a], [MGP09b]. Er zerlegt dabei die verändernden Aktivitäten so wie Löschen, Anlegen, Ändern von Objekten, die sich auf eine oder mehrere Komponenten beziehen, in kleinen elementaren Teilschritten. Der Ansatz zur Automatisierung besteht in der Annahme, dass man alle sich während der Modellierung ergebenden Arbeitsschritte wie folgt in vier Veränderungstypen zusammenfasst:

- ADD(Objekt)

- DEL(Objekt)
- PREMOD(Vorzustand des Objektes)
- POSTMOD(Nachzustand des Objektes)

Bei den Informationen handelt es sich hier im Wesentlichen um die Eigenschaften sowie die Elementtypen zu diesen beobachteten und geänderten Elementen. Danach werden die Information im XMI-Format gespeichert, da diese nicht nur von Enterprise Architect geöffnet werden können, sondern auch die Nutzer darauf zugreifen können. [Kus07].

4.4. Tracestore

Um eine automatische Linkpflege zu realisieren, muss das System in der Lage sein, die Traceability Informationen eigenständig zu verwalten [Kit08]. Es muss neben dem Anlegen und Löschen der Links auch eine Anfrage der Traceability-Link-Information ermöglichen. Dazu stehen zwei verschiedene Datenbankstrukturen zur Verfügung. Man kann die Traceability-Link-Beziehungen in einer eigenen Datenbankstruktur TraceStore, die wiederum ein Add-IN des Modellierungstools darstellt, ablegen oder in einen externen Repository sowie EXTESSY ToolNet speichern [MGP09a], [MGP09b]. In dieser Arbeit wird die erste Variante vorgestellt. Die Kommunikation zwischen RuleEngine und TraceStore erfolgt hierbei direkt. Aber dafür verlangt RuleEngine eine Schnittstelle, über die es Traceability-Links anfragen und aktualisieren kann. Die Realisierung der Ablage innerhalb einer Datenbankstruktur hat bezüglich der automatisierten Verarbeitung einen entscheidenden Vorteil. Mit ihr kann man die abgespeicherten Werte einer Verbindung in der Datenbank bewahren, auch wenn es die betroffenen Elemente im Modell nicht mehr gibt. Die Hauptidee ist die Speicherung der Traceability-Informationen bis zur Anforderung einer separaten Löschung. Das wurde dadurch erreicht, dass eine Art History in Form eines Zwischenspeichers eingeführt wurde [Kit08]. Die Abspeicherung erfolgt in einem gesondert festgelegten Paket innerhalb einer eigenen Modellumgebung in einem Projekt von Enterprise Architect. Die Ablage innerhalb eines eigenen Paketes hat den positiven Effekt, dass sich automatisch ein Diagramm erstellen lässt, auf dem die Verkettung der Objekte visualisiert werden kann. Somit wurde es ermöglicht, Traceability-Relationen in grafischer Form darzustellen.

4.5. Verfeinerte Problemstellung

Der TraceMaintainer stellt eine softwaretechnische Realisierung der automatischen bzw. teilautomatischen Nachführung von Relationsänderungen zwischen Analyse- und Designobjekten dar [MGP09a]. Der TraceMaintainer ist ein sehr hilfreiches Tool. Mit dessen Hilfe sind die Verkettungen für eine lückenlose Verfolgbarkeit zu protokollieren, sodass man sehr leicht alle beteiligten Komponenten einer SW-Anforderung erkennt, indem man einfach die gesetzten Verbindungen bzw. Traceability-Links nachvollzieht [GF94]. Dadurch lassen sich potentielle Fehler einfach finden und gezielt beseitigen. Durch die Traceability-Links bzw. mit deren Hilfe können die Verifikationen und/oder die Validierung einer SW-Anforderung entscheidend ver-

bessert werden [Kit08].
Obwohl der TraceMaintainer eine große Lücke bei SW-Entwicklung und Modellierung füllt [Kus07], [MGP09a], bedeutet dies noch längst nicht, dass es ein perfektes Tool ist. Einer seiner Nachteile ist, dass er nur lokal auf einem PC angewendet werden kann. Das führt wiederum dazu, dass man auf einem SW-Modell allein arbeiten muss. Also es ist zur gleichen Zeit keine Teamarbeit auf einem SW-Modell unter Verwendung von TraceMaintainer möglich. Im Gegensatz dazu stehen die Unternehmensstrukturen global agierender SW-Firmen, welche ein großes Interesse daran haben, dass ihre Mitarbeiter räumlich verteilt an denselben Dokumenten arbeiten können. Ein Softwaresystem, das diese Art der Arbeit unterstützt, muss deshalb mit verteilten Datenbeständen arbeiten können und muss auf verschiedene Organisationsstrukturen anpassbar sein [KCH04]. Wenn ein Softwaresystem wie TraceMaintainer keine Teamarbeit in den räumlich verteilten Umgebungen unterstützt, macht es wenig Sinn, ein solches Tool zu entwickeln.
Kurz gesagt, wenn der TraceMaintainer über Rechnergrenzen hinweg anwendbar wäre, dann könnte man bequem und leicht eine Teamarbeit auf einem SW-Modell an völlig verschiedenen Orten im Netzwerk ermöglichen. Deswegen ist es jetzt höchste Zeit, den TraceMaintainer an die Bedürfnisse des Marktes anzupassen, damit er noch breiter eingesetzt werden kann.

5. RemotingHub als Middleware

Es wurden in dem letzten Kapitel die TraceMaintainer und dessen Komponenten vorgestellt. Er besteht aus mindestens 4 Teilen sowie einem Modellierungstool, Eventgenerator (EG), RuleEngine (RE), TraceStore (TS) , die miteinander unbedingt zusammenarbeiten müssen. Da ein EG unbedingt auf einem Modellierungstool arbeiten und die Events für RE erzeugen muss, stellt er zusammen mit einem Modellierungstool eine aktive Komponente dar. Dagegen sind RE und TS passive Komponenten. An dieser Stelle ist es besser zu erwähnen, dass man hier davon ausgeht, alle dieser SW-Komponenten seien vorhanden. Zur Zeit können alle diese Komponenten nur zum lokalen Betrieb angewendet werden. In diesem Sinne ist die TM-Anwendung als eine monolithische Anwendung und nicht als eine verteilte Anwendung zu betrachten. Aus diesem Grund (siehe den Abschnitt) muss der TM in einer verteilten Umgebung funktionstüchtig gemacht werden. Dieses Kapitel beschäftigt sich mit den konzeptionellen Ansätzen, die den TM dazu führt.

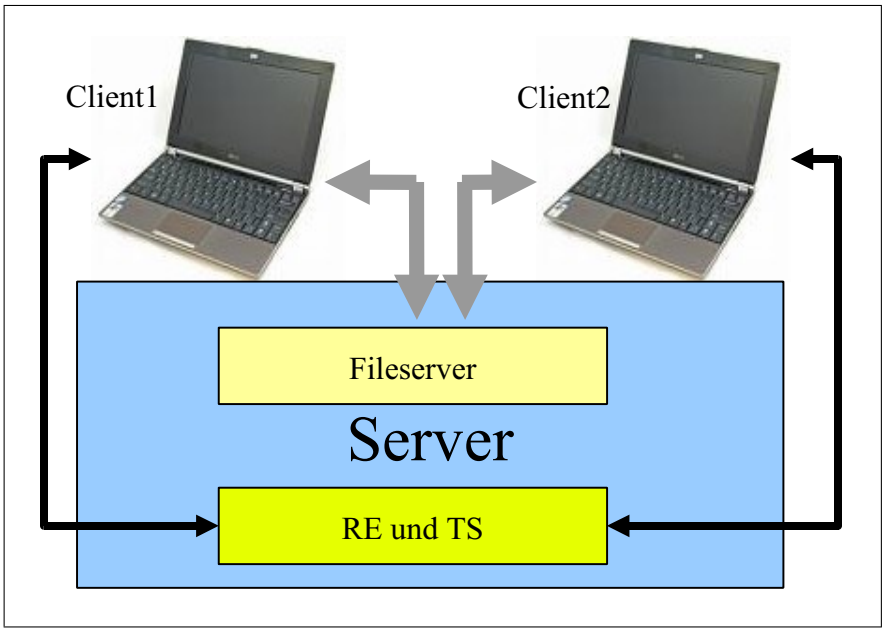

Abbildung 5.1.: Ein Anwendungsbeispiel für traceMAINTAINER

5.1. Ein kleines Anwendungsbeispiel

In der bisherigen TM-Anwendung können die Komponenten im einer Anwendungsdomäne bei lokalem Betrieb miteinander ohne große Schwierigkeiten direkt kommunizieren. Aber diese Art der Anwendung von TraceMaintainer kann bei heutigen modernen SW-Entwicklungsarbeiten unter Umständen wenig Sinn machen. Es kann sein, dass mehrere SW-Entwickler auf die Idee kommen, dass sie von verschiedenen Rechnern im Netzwerk aus an einem gemeinsamen SW-Modell arbeiten wollen. Anhand von Abbildung 5.1 könnte man sich das beschriebene Anwendungsszenario vorstellen. Angenommen, die zwei verschiedenen SW-Entwickler arbeiten an einem Projekt in räumlich getrennten Orten und möchten über die Verifikation und Validierung der SW-Anforderung ihres Projektes unter Verwendung von TraceMaintainer sprechen. Damit ihre Diskussion ohne weitere Missverständnisse verläuft, müssen sie die gleichen Traceability-Links von dem aktuellen Modell bzw. Projekt auf beiden Seiten anwenden können. Dies klingt recht einfach, aber wie macht man das? Dazu werden in der Regel viele verschiedene Lösungen angeboten. Eine davon ist, wie es Abbildung 5.1 zeigt, einen dritten Kommunikationspartner, d.h. einen Serverrechner anzuwenden, sodass in diesem Fall die RE und TS auf diesem Server den anderen Komponenten immer verfügbar bleiben können.
Um die Traceability-Links von dem aktuellen Modell von beiden Seiten verfügbar zu machen, müssen die Modellierungstools der beiden Anwender in diesem Szenario mit den RE und TS auf dem Serverrechner eine Kommunikation aufbauen. Die Gegebenheit, dass die beiden Anwender sich an völlig verschiedenen Orten im Netzwerk befinden, macht den Aufbau der Kommunikation nicht so leicht, wie man dies von monolithischen Anwendungen auf einem Rechner gewohnt ist. Die Methodenaufrufe in diesen Fällen sind im Vergleich zur lokalen Kommunikation relativ komplizierte Vorgänge. In diesem Zusammenhang spricht man von einem entfernten Methodenaufruf, einem Remote Procedure Call (RPC) [siehe Abschnitt 3.1]. In diesem Fall ist ein Zugriff auf einen gemeinsamen Speicherbereich, wie beispielsweise den Stack, gar nicht möglich. Deshalb werden entfernte Methodenaufrufe durch zwei spezielle Systemprozeduren durchgeführt. Kurz gesagt ist dafür ein spezieller Mechanismus des Nachrichtentransports nötig, der die Aufgabe übernimmt, Nachrichten von einer bestimmten Komponente zu einer anderen Komponente zu verteilen bzw. zu übertragen.
In diesem Kapitel wird über die Realisierungsmöglichkeiten eines solchen Kommunikationsverfahrens konzeptionell diskutiert. Zunächst sollen die Anforderungen an das zu realisierende System angerissen und anschließend einige Designs skizziert werden. Man muss wissen, dass es sicherlich viele Wege gibt, die zum Ziel führen können und man kann lange über eine vernünftige Lösung diskutieren. Jedoch wurde das hier vorzustellende Vorgehen als geeignet für die Problemlösung befunden.

5.2. RemotingHub ist ein SW-Hub

In Einzelfällen kann eine verteilte Anwendung sowohl in verschiedenen Prozessen als auch auf einem Rechner ausgeführt werden [TS03], [Ill07]. Neben der eigentlichen Anwendung, die auf mehrere Prozesse bzw. Rechner aufgeteilt wird, sind deshalb weitere Softwarekomponenten notwendig. Diese werden als Middleware [TS03] bezeichnet. Die Middleware realisiert

die Kommunikation der Anwendung mit dem darunter liegenden Betriebssystem des Rechners. In Abbildung 2.1 ist das entsprechende Modell zu sehen. Genauso soll das RemotingHub auch die Rolle einer Middleware spielen. Es muss mit Hilfe eines effizienten Verfahrens eine weitere SW-Komponente gebaut werden, die sowohl die Kommunikation der TraceMaintainer-Komponente in der verteilten Umgebung ermöglicht als auch die ganze Nachrichtenverteilung der SW-Komponente des TraceMaintainer koordiniert.
Das RemotingHub ist nach dem Prinzip eines Netzwerk-Hubs (siehe im Abschnitt 2.4.1) entworfen. Während ein Netzwerk-Hub Hardware ist, sollte RemotingHub Software sein, die noch einige zusätzliche spezielle Funktionalitäten enthalten kann, die ein Netzwerk-Hub nicht besitzt. Bei Einsatz eines Hubs im Netz wird durch die Verkabelung im physikalischen Sinne eine Stern-Topologie realisiert. Der logische Aufbau ähnelt dem einer Bus-Topologie, weil jede gesendete Information alle Teilnehmer erreicht. Genau dieses Grundkonzept enthält das RemotingHub auch und sollte außerdem noch weitere Funktionalitäten anbieten. Dazu gehören z.B. die gezielte Verteilung von Nachrichten (Multicast), die Verwaltung der SW-Komponente, welche daran angeschlossen bzw. damit verbunden sind, die Verarbeitung und Verwaltung von Nachrichten, Sessionverwaltung, usw. RemotingHub soll in dieser verteilten TM-Anwendung die zentrale Rolle spielen und die SW-Komponenten des TraceMaintainer sollen rund um die Zentrale arbeiten. In dieser verteilten TM-Anwendung sollen jedoch alle diese TM-zugehörigen SW-Komponenten komplett unabhängig voneinander arbeiten.

5.3. Systemanforderungen

Die grundsätzlichen Funktionalitäten des Nachrichtentransportsystems kann sich jeder sicherlich relativ einfach , das realisiert werden muss, vorstellen. Es wurde auch in den vergangen Kapiteln mehrfach erwähnt, dass es hier um die Übermittlung von Nachrichten von Komponente A nach Komponente B geht. Wie bereits in den vergangenen Kapiteln herausgearbeitet wurde, ist nun der Nachrichtenverkehr zwischen SW-Komponenten des TraceMaintainer auf einem lokalen Rechner nicht so sehr interessant. Von weitaus größerem Interesse wäre hier die Intermaschinen-Kommunikation, die überall im Netzwerk ihre Anwendung findet, und die Art der Übermittlung der Nachrichten zwischen den Clients untereinander. Die Aufgabenstellung erfordert die Realisierung eines Kommunikationsmechanismus, der mehrere eigenständige SW-Komponenten miteinander bei deren lokaler und entfernter Kommunikation koordinieren kann. Dieser Umstand verlangt jedoch nicht nur die Implementierung einer Interprozesskommunikation, sondern auch Intermaschinen-Kommunikation, welche einen gesteigerten Aufwand zur Folge hätte. Auch ist es für den Modellierer relativ uninteressant zu wissen, wer eine bestimmte Nachricht sendet und wer eine Nachricht bekommt. Es ergeben sich immer wieder neue Probleme bei der Integration der eigenen Kommunikationsinfrastruktur ins Betriebssystem. Des Weiteren entstehen programmiertechnische Hindernisse.
Das Hauptaugenmerk liegt aber auf der Realisierung einer eigenständigen SW-Komponente zum TraceMaintainer. Um die Traceability-Link-Informationen über Rechnergrenzen hinweg verfügbar zu machen, muss das System in der Lage sein, die lokale und entfernte Netzwerk-Kommunikation zu unterstützen. Es muss neben der Übermittlung der Nachrichten von Komponente A nach Komponente B auch ein relativ intelligentes Verhalten bei der Nachrichtenüber-

tragung sowie z.B. ein zuverlässiger Nachrichtentransport an das System implementiert sein. Die Nachrichtenübermittlung zwischen zwei Kommunikationspartnern soll, egal ob sie sich auf einem Rechner oder auf verschiedenen Rechnern im Internet befinden, möglichst schnell und asynchron erfolgen. Um keine gravierenden Änderungen an den bestehenden Komponenten durchführen zu müssen, sollte die entfernte Kommunikation bzw. Nachrichtentransport in der bereits umgesetzten Art und Weise (genauso wie bei lokaler Nachrichtenübermittlung) erfolgen.
Neben der Realisierung des Nachrichtentransports hat das hier zu konzipierende System noch einige Aufgaben zur Laufzeit zu erledigen. Nach bestimmten vordefinierten Regeln müssen bestimmte passive SW-Komponenten sowie RE, TS (siehe Abschnitt 5) starten und schließen können. Diese Systemfähigkeit verursacht noch weitere Aufgabenstellungen sowie Sessionverwaltung, die vollautomatisch vom System erledigt werden muss. Nähere Informationen sind dem nächsten Kapitel zu entnehmen.
Hier sind die Anforderungen zusammengefasst aufgelistet:

- Das System soll die Kommunikationsinfrastruktur für die SW-Komponenten des TraceMaintainer bereitstellen.
- Das System soll die Sessions mit bestimmten Clients aufbauen und sie verwalten können, wobei der Aufbau einer Session durch einen bestimmten Tool-Adapter sowie EG getriggert wird. Die Sessionverwaltung soll vom Benutzer nicht beeinflusst, sondern vom System komplett selbständig erledigt werden.
- Der Arbeitsablauf des Systems soll weitgehend autonom im Hintergrund stattfinden, um die Arbeit des Entwicklers nicht zu behindern. Dafür soll z.B. einen Windowsdienst gebaut werden.
- Das System soll noch in der Lage sein, bestimmte Clients nach bestimmten Registrierungsregeln starten und schließen zu können. Dafür soll ein Regelkatalog entworfen werden, der erweiterbar sein soll.
- Das System soll mit Client-Nachrichten asynchron umgehen können. Dabei soll das System den zuverlässigen Datentransport zwischen den Clients gewährleisten.
- Das System soll mit einer modernen und geeigneten Softwaretechnik realisiert werden. Deswegen ist neben diesen zu realisierenden Systemfunktionalitäten erforderlich, noch über den Stand der Technik zu recherchieren, um einen geeigneten Prototyp zu entwickeln. Wobei die eingesetzte Technologie dafür sorgen muss, die Kommunikation zwischen SW-Komponenten des TraceMaintainer möglichst schnell erfolgen zu lassen.

Zudem wird eine Reihe nicht funktionaler Anforderungen gestellt, die sich im wesentlichen aus Erweiterbarkeit und Wiederverwendbarkeit des Ansatzes ergeben.

- Zur Entkopplung des Systems müssen eine oder mehrere Schnittstellen realisiert werden, über die das Empfangen der Nachrichten von SW-Komponenten des TraceMaintainer läuft und die Antworten auf diese Nachrichten weitergeleitet wird.
- Der Datenteil des Systems, welche die erstellten Regeln zum Start- bzw. Stop-Verhalten von anderen SW-Komponenten sowie die Konfigurationsdatei umfasst, soll veränderbar

und korrigierbar, besonders anpassbar für zukünftige Anwendungen sein.

5.4. Auswahl eines geeigneten Architekturmodells

Es wurde im Abschnitt 5.1 ein mögliches Szenario für die verteilte TM-Anwendung gegeben und sehr kurz über die Kommunikationsmöglichkeiten ihrer Komponenten diskutiert. Trotzdem bleibt die Frage: Wie können die einzelnen Teile der verteilten TM-Anwendung in eine logische Beziehung zueinander gebracht werden? Ein grundlegendes Modell zur Verdeutlichung dieser Frage ist, wie bereits in Kapitel 2 beschrieben wurde, das Client-Server-Modell (CS-Modell) oder Peer-to-Peer Modell (PTP-Modell).

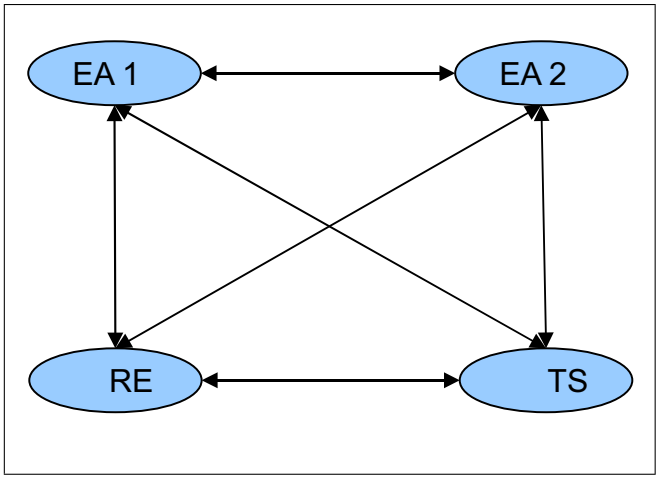

Abbildung 5.2.: Ein verteilte TM-Anwendung im PTP-Modell

Da im PTP-Modell alle Peers (Oberbegriff für Clients und Servers) gleichberechtigt sind bzw. sein müssen, bedeutet es in einer verteilten TraceMaintainer-Anwendung mehr Aufwand bei der Realisierung ihrer Kommunikation. Dazu sollte jede SW-Komponente einen eigenen Mechanismus (z.B ein Server) besitzen, um ihre Services anzubieten. Außerdem muss sie sich auch über die Services von anderen SW-Komponenten informieren können. Allein das kostet viele Rechnerressourcen. Dazu kommt noch eine komplizierte Sessionbildung und deren Verwaltung. Wie bereits beschrieben, besteht eine minimale Session in einer TM-Anwendung aus einem Modellierungtool sowie RuleEngine und TraceStore, wobei angenommen werden kann, dass Eventgenerator (EG) als AddIn in dem Modellierungstool vorhanden ist. Das ist Mindestvoraussetzung dafür, dass man allein das Link-Nachführungssystem TraceMaintainer benutzen kann. Die TM-Anwendung besteht mindestens aus drei SW-Komponenten (wenn man EG und EA zusammenzählt).
Wenn das Tool in einer verteilten PTP-Umgebung angewendet würde, sollte dann ein vollständiger Graph zur seiner Kommunikation entstehen. Wie Abbildung 5.2 zeigt, kann es sein, dass zwei Entwickler im Netz mit Hilfe ihrer Modellierungstools Enterprise Architect (EA) zueinander in Verbindung stehen. Außer ihrer Kommunikation zueinander sollten sie sich jeweils

mit Hilfe von RuleEngine (RE) und TraceStore (TS) eine bidirektionale Verbindung aufbauen. Dazu kommt noch, dass jedes Tool zu den beiden Komponenten RE und TS verbinden muss. Das sind alles direkte Kommunikationsbeziehungen zueinander. Unter einer solchen direkten Kommunikation versteht man hier, dass ein Peer (z.B. Source-Client) genau weiß, mit welchem Peer (Ziel-Clients bzw. ihre Kommunikationspartner) er zu tun hat, und an wen er eine Nachricht verschickt. Der Ziel-Client verarbeitet auch die empfangene Nachricht und sendet ihre Antwort zurück. Dafür muss der Source-Client eine Nachricht an dessen Ziel-Client nach und nach zusenden und dann am Ende die Antworten selbst verarbeiten. Ungeachtet des großen Implementierungsaufwands sind die Kosten einer solchen direkten Kommunikations sehr hoch. Hinzu kommt noch, dass jeder sich allein um die Verarbeitung der Nachrichten kümmern muss. Das ist sehr umständlich und kompliziert. Deswegen sind PTP-Modelle für die verteilte TM-Anwendung eher ungeeignet.

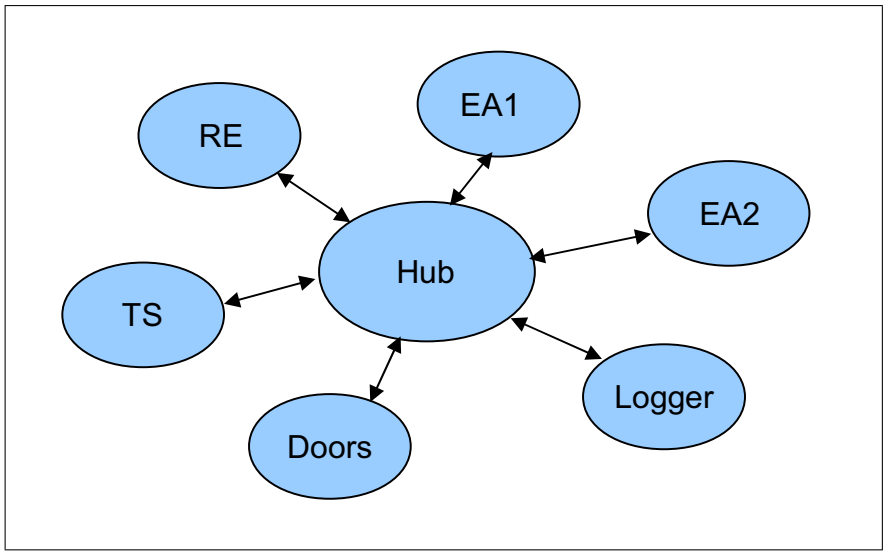

Abbildung 5.3.: CS-Modell für die verteilten TraceMaintainer-Anwendung

In einem CS-Modell sieht die ganze Problematik anders aus. Zunächst werden die Rollen der jeweiligen Anwendungen festgelegt. In diesem Zusammenhang werden in dieser Arbeit alle Komponenten des TraceMaintainer als Client und das hier zu konzipierende Nachrichtentransportsystem, RemotingHub, als Server betrachtet. Somit ist klar, dass die SW-Komponenten des TraceMaintainer nur die angebotenen Services von RemotingHub nutzen müssen, um ihre Kommunikationsziele zu erreichen. Topologisch stehen dann die Kommunikationspartner von TraceMaintainer in einem Ring. Das macht nicht nur ihre Kommunikation einfacher, sondern es entfällt auch der Aufwand der Nachrichtenverwaltung von Clients. Somit werden eine benutzerfreundliche Interprozess-Kommunikation und Intermaschinen-Kommunikation ermöglicht. An der Stelle soll noch einmal daran erinnert werden, dass diese Komponente als gegeben angenommen wird.
Wie aus Abbildung 5.3 ersichtlich ist, spielt das RemotingHub die Rolle eines Servers. Modellierungstools wie Enterprise Architect und DOORS (ein Tool für Anforderungsmanagement

von IBM für Software) melden sich beim Server mit ihrem Client-Mechanismen und bauen somit die bidirektionale Kommunikation zu ihren Kommunikationspartnern auf. Dazu sollte der Server entsprechende Schnittstellen bereitstellen. Auf dieses Thema wird im kommenden Kapitel detailliert eingegangen.

5.5. Welche SW-Technologie passt am besten?

In dem Kapitel 3 wurde jeweils ein kurzer Überblick über die aktuellen Technologien für die verteilten Systeme bzw. Anwendungen gegeben. Die Frage ist, welche von diesen Technologien zur Realisierung des Prototyps eingesetzt wird. In [KCH04] wurde die Eignung dieser Technologien für drei unterschiedliche Anwendungsszenarien untersucht. In Tabelle 5.1 sind die Gegenüberstellungen zu sehen, wobei an dieser Stelle nur ausgewählte Szenarien aufgeführt werden. Die drei unterschiedlichen Anwendungsszenarien sind:

- Ein Client-Server und Peer-to-Peer-Anwendungen sind in kleineren Netzen zu betrachten; z.B. ein Server mit mehreren Clients, die reine Dienstnutzer sind und keine eigene Dienste anbieten.
- Anwendungen im Intranet; in diesem Fall sind die Unternehmensnetze zu betrachten.
- Internetanwendungen, B2B, EAI; sobald das Internet ins Spiel kommt, ändern sich viele Eigenschaften des Systems.

Szenario	Remoting	Webservices	WCF
C/S, P2P	sehr gut	gut	befriedigend
Intranet	durchschnitt	gut	sehr gut
Web	schlecht	sehr gut	sehr gut

Tabelle 5.1.: Eignung verschiedene Technologien für Anwendungsszenarien

Das Untersuchungsergebnis zeigt, dass .Net Remoting eine sehr gute Wahl ist, wenn eine schnelle CLR-basierte Kommunikation (CLR-to-CLR) benötigt wird. Nach langer Recherche zu dieser Diplomarbeit über diese Technologien wurde schließlich entschieden, für die prototypische Realisierung des vorgestellten Konzepts .Net Remoting einzusetzen. Die Gründe hierfür sind:

- Weil die hier zu konzipierende verteilte TM-Anwendung noch eine relativ kleine Client-Server-Anwendung ist.
- Da Kommunikationspartner während der Laufzeit einen hohen Anspruch an Kommunikationsgeschwindigkeit haben.
- Die eigene Erfahrung mit .Net Remoting spielt auch eine wichtige Rolle bei der Entscheidung.

Die Wahl fiel auch auf die .Net Remoting Technologie aufgrund deren immer größer werdenden Verbreitung und der damit verbundenen einfacheren Integrationsfähigkeit und Anpassbarkeit an andere relevante Systeme. Im Vordergrund stand auch die Erweiterbarkeit und Flexibilität der

Anwendung [KCH04], um sie möglichst einfach von einem bestehenden System in ein neues, sich zur Zeit noch in der Entwicklung befindenden System überführen zu können. Außerdem muss erwähnt werden, warum nur unter diesen .Net-Technologien eine Auswahl getroffen muss. Der Grund dafür ist, dass der TraceMaintainer zum wesentlichen Teil in .Net entwickelt worden ist und man somit auch von einer bestmöglichen Kompatibilität ausgehen kann.
Im Allgemeinen ist es relativ schwierig, über all diese Technologien tiefgehend zu recherchieren und dann stahlfest von einer Technik überzeugt zu sein, dass man es für die einzig passende Technologie zur Realisierung eines Prototyps, RemotingHubs hält. Das Recherche-Ergebnis zeigt, dass es keine eindeutige Aussage darüber geben kann, immer eine bestimmte Technologie für eine bestimmte Anwendung einzusetzen. Die technologischen Auswahlkriterien von früheren Entwicklern lassen sich häufig auf allgemein bekannte Fälle begrenzen. In den meisten Fällen entscheidet sich der Entwickler allein bezogen auf die eigene Problematik objektiv und pragmatisch für eine bestimmte Technologie.

5.6. Vorgehen

Es wurde im letzten Kapitel klar gemacht, dass der TraceMaintainer-Anwendung aus verschiedenen Komponenten besteht. In Abschnitt 5.1 wurde ein potentielles verteiltes Anwendungsbeispiel dafür vorgestellt. Neben der eigentlichen Anwendung, die auf mehrere Prozesse bzw. Rechner aufgeteilt wird, sind weitere Softwarekomponenten notwendig. Diese werden insgesamt als RemotingHub bezeichnet.
Gemäß den Anforderungen in Abschnitt 5.3 kann das RemotingHub funktional unter anderem in RemotingHubMediator und RemotingHubService unterteilt werden. Darunter ist der Aufbau eines RemotingHubMediator der wichtigste Schritt und die Voraussetzung dafür, dass der letzte Teil ohne weitere Schwerigkeiten funktionieren kann. Hier wird zunächst mit Hilfe der eingesetzten Technologie die Kommunikationsinfrastruktur zur Herstellung der Verbindung bereitgestellt und Kommunikationskanäle zwischen Server und Clients initialisiert. Danach können sie problemlos zueinander Nachrichten senden. In diesem Sinne spielt der RemotingHubMediator die Rolle einer Middleware (siehe den Abschnitt 2.1), welche die Kommunikation der jeweiligen Anwendung mit dem darunter liegenden Betriebssystem des Rechners realisiert. So zu sagen, ist der RemotingHubMediator ist ein kleine problemspezifische Middleware für die verteilte TM-Anwendung, sodass sie die ganze Kommunikation zwischen den SW-Komponenten der TM-Anwendung während der Laufzeit in einer verteilten Umgebung ermöglicht.
RemotingHubService ist eine wichtige Komponente des Systems. Während der RemotingHubMediator die Grundlage für die Kommunikation ermöglicht, hat der RemotingHubService eine Reihe weitere Systemaufgaben (siehe Abschnitt 5.3) zu erledigen. Alle diese Aufgaben richten sich ach dem Verhalten der Clients. Deswegen muss zwischen folgenden vier verschiedenen Phasen unterschieden werden, um das Systemverhalten übersichtlich zu beschreiben:

1. Client meldet sich an —> Server meldet Client an
2. Client arbeitet —> Server arbeitet
3. Client hat Fehler oder stirbt —> Server reagiert entsprechend darauf
4. Client meldet sich ab —> Server meldet Client ab.

Diese vier Fälle teilen die Aufgaben von Server in zwei Unterteile:

- Die Sessionverwaltung: Der An-und Abmeldungsmechanismus ist zuständig für die Fälle 1 und 4. Das System muss bei An- und Abmeldung eines aktiven Clients die passiven Clients an- und abmelden können und dafür in der Lage sein, zur Laufzeit die passiven Clients zu starten. Mit Hilfe dieser Systemfunktionalität ist möglich, mit bestimmten Clients Sessions aufzubauen und wieder zu zerstören.
- Nachrichtentransport des Systems: Dies ist Kernaufgabe des Systems. Dazu gehören die Nachrichten bzw. Clients zu verwalten. Dies betrifft die Fälle 2 und 3.

Im Rahmen dieser Arbeit wird nur der Entwurf des Nachrichtentransportsystems, RemotingHub betrachtet, da dieser die Umsetzung des Ansatzes darstellt. Die notwendigen Komponenten wie RE und TS, die dieses System benutzen, werden als vorhanden angenommen. Es werden entsprechende Schnittstellen zur Ankopplung dieser Komponenten an das System definiert. Im Folgenden wird das Nachrichtentransportsystem detailliert vorgestellt.

5.7. Aufbau der Kommunikationsinfrastruktur

Nach einer Recherche über den aktuellen Stand der Technik zur Realisierung einer verteilten Anwendung (siehe Kapitel 3) muss eine modernere und geeignetere Technologie ausgewählt werden (siehe Abschnitt 5.5). Bei dieser Technologieauswahl sollte man die Kommunikationsgeschwindigkeit zwischen den SW-Komponenten des TraceMaintainer als wichtiges Kriterium in Betracht ziehen. Die zu realisierende Kommunikationsinfrastruktur unterstützt nicht nur Interprozesskommunikation und Intermaschinen-Kommunikation, sondern muss auch in der Lage sein, automatisch auszuwählen, welche von den Kommunikationsvarianten der aktuell anzumeldende Client bzw. SW-Komponente des TraceMaintainer benötigt wird. Der Vorteile dieser Vorgehensweise ist, dass eine schnellstmögliche, optimale Kommunikation zwischen Clients angeboten wird und sich der Entwickler darüber keine Sorgen mehr machen muss.
Die Kommunikationspartner im CS-Modell sind nicht gleichberechtigt, sie haben in der Regel schon fest verteilte Rollen. Dieses Verhalten verursacht wiederum aus der Sicht der Clients ein neues Problem beim Erhalt einer Rückantwort, wenn ein Client eine Anfrage an den Server sendet. Im Folgenden werden zwei problemspezifische Lösungen vorgestellt, wobei diese nach einer Technologieauswahl entworfen und praktisch getestet wurden.

5.7.1. Erste Lösung mit Polling

Es wird in einer Schleife permanent abgefragt, ob die asynchrone Operation bereits beendet ist. Dieses Verfahren wird als Polling bezeichnet [Com]. Die herkömmliche Art der Lösung zum Nachrichtenrücktransport an den jeweiligen Klienten war mit Polling, da hierdurch die bidirektionale Kommunikation zwischen Server und Clients im System ermöglicht wird. Wird der Server in bestimmten Zeitabständen im Polling-Verfahren vom Client angeklopft, wenn dieser davor mindestens eine Nachricht an den Server geschickt hat, ob eine Antwort dafür vorliegt

[TS03]. Das steht bezüglich des Aufwands in keinem Verhältnis, da jede Abfrage-/Antwortfolge auch bestimmte Ressourcen kostet. Abbildung 5.4 zeigt diese Art des Systementwurfs im RemotingHub, sodass jeder Client eine Nachricht an den Server ohne Problem senden kann. Das Problem ist dabei, wie schon erwähnt, beim Erhalt bestimmter Nachrichten vom Server. In diesem Verfahren sollte der Client sich selbst darum kümmern, seine Messages selbständig aus dem Server zu holen, in dem er regelmäßig den Server nach Nachrichten, die dem Client zugestellt wurden, abfragt. Die Annahme dafür ist, dass der Server eine Message von einem bestimmten Client nach bestimmtem Messageausgang weiterleitet und in lokalen Warteschlangen beim Server abgelegt wird, welche der Client währenddessen abfragen muss. Der Nachteil dieses Verfahrens ist der Abstand der Abfragen nach Client-Nachrichten beim Server. Eine Abfrage mit kleinerem Zeitabstand ist zwar gut für die schnellere Kommunikation, aber der Server wird ausgelastet. Eine Abfrage mit größerem Zeitabstand hat den umgekehrten Effekt. Deswegen wurde dieses Verfahren für die Realisierung einer verteilten Kommunikation für das System als ungeeignet befunden.

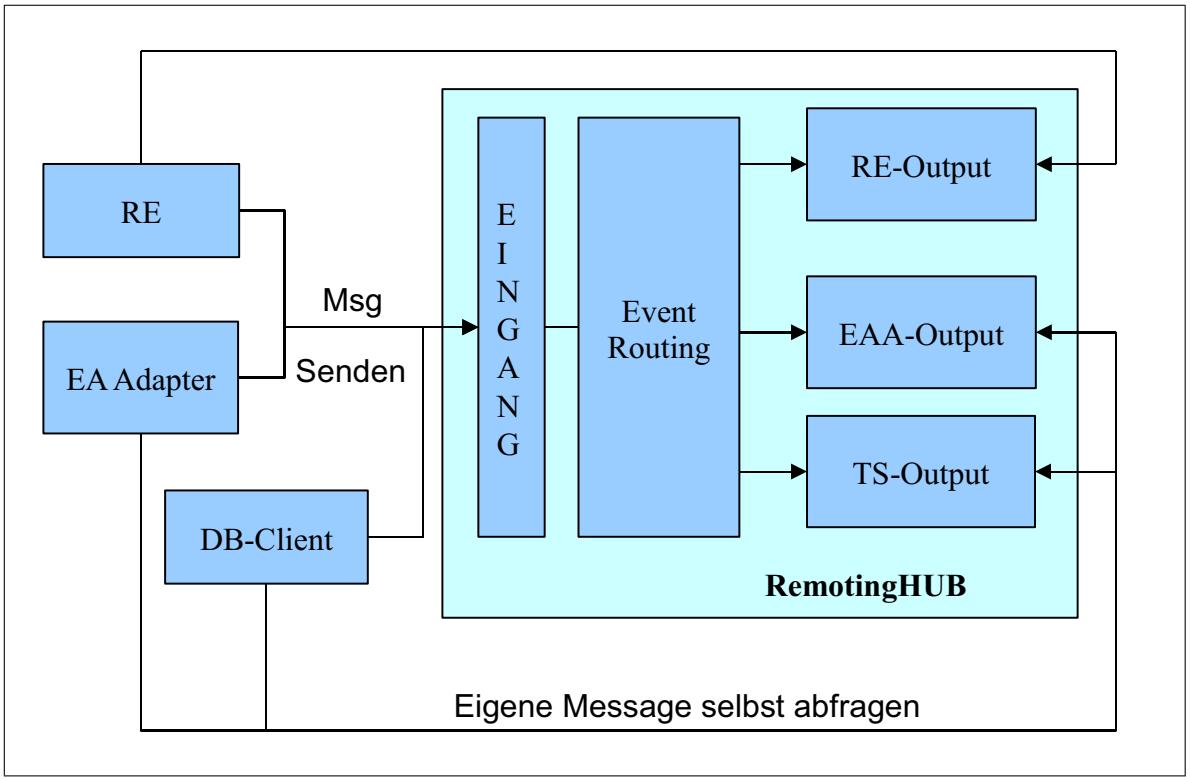

Abbildung 5.4.: Polling in RemotingHub

5.7.2. Callback ist zweite Lösung

Tannenbaum hat in [TS03] das *Callback-Modell* wie folgt beschrieben:

> „Im *Callback*-Modell stellt ein Client ein Objekt bereit, das eine Schnittstelle implementiert, welche *Callback-Methoden* enthält. Diese Methoden können von dem zugrunde liegenden Kommunikationssystem aufgerufen werden, um das Ergebnis eines asynchronen Aufrufs zu übergeben."

Es wird eine Rückrufmethode (*Callback*-Methode) definiert, die aufgerufen wird, sobald das Ergebnis vorliegt [Com]. *Callback* ist ein alternatives Vorgehen für den Erhalt einer Rückantwort, sodass der Client einen Rückruf bei seiner Initialisierung registriert. Die Rückrufe können dann Funktionseingangspunkte oder Ereignisse sein. Beim Eintreffen der Rückantwort werden dann die registrierten Funktionen bzw. Ereignisbehandlungsroutinen aktiviert [MBW08]. Wie in Abbildung 5.5 gezeigt, kann der Client nach dem Server direkt eine Message über ein Remoteobjekt senden. Der eigentliche kritische Punkt, der Erhalt einer Rückantwort, ist in diesem Verfahren mit einem kleinen Trick gelöst. Dieser besteht darin, dass der Client vor seiner Registrierung beim Server auch ein remotefähiges Objekt erzeugt und dem Server übergibt. Aber davor muss es sich zu diesem übergebenen Objekt ankern. Normalerweise stellt dieses Remoteobjekt eine Ereignisroutine frei zur Verfügung, in die der Client anhakt, damit er sich die Messages des Servers anhört. Eine solche *Callback*-Klasse sieht wie folgt aus:

```
public class CallbackSink:MarshalByRefObject
 {
      public event delEventInfo OnServerToClient;
       [OneWay]
       public void HandleToClient(Event info )
        {
          if (OnServerToClient != null)
            {
              OnServerToClient(info );
            };
        }
 }
```

Während der Registrierung registriert der Client einen *Callback-Methode* zu Ereignisbehandlungsroutine *OnServerToClient* innerhalb dieser Instanz des Remoteobjekts wie folgt:

```
CallbackSink callbackSink = new CallbackSink();
callbackSink.OnServerToClient += new delEventInfo(CallbackSink_OnHostToClient);
```

Somit hat der Server eine gültige Methode eines Remoteobjekts, das auf dem Client lebt. Da der Server auf die *Callback*-Nachrichten keine Rückantwort erwartet, sollte es genügen, wenn das Remoteobjekt für *Callback* eine einfache Kommunikation (One-Way-Kommunikation) ist, d.h. der Server schickt nur eine Nachricht ab und kümmert sich dann nicht mehr um eine Rückantwort. Wenn eine Rückantwort nötig sein sollte, könnte der Client diese auf normalem Wege beantworten. Dieses *Callback*-Verfahren wird nur dann funktionieren, wenn der Server über den Remoteobjekttyp CallbackSink Bescheid weiß. Der Vorteil dieses Verfahrens ist, dass Clients und Server direkt miteinander ohne großen Aufwand kommunizieren können. Im Vergleich zum Polling-Verfahren hat man also keine Verzögerung bzw. Serverauslastung bei der Kommunikation.

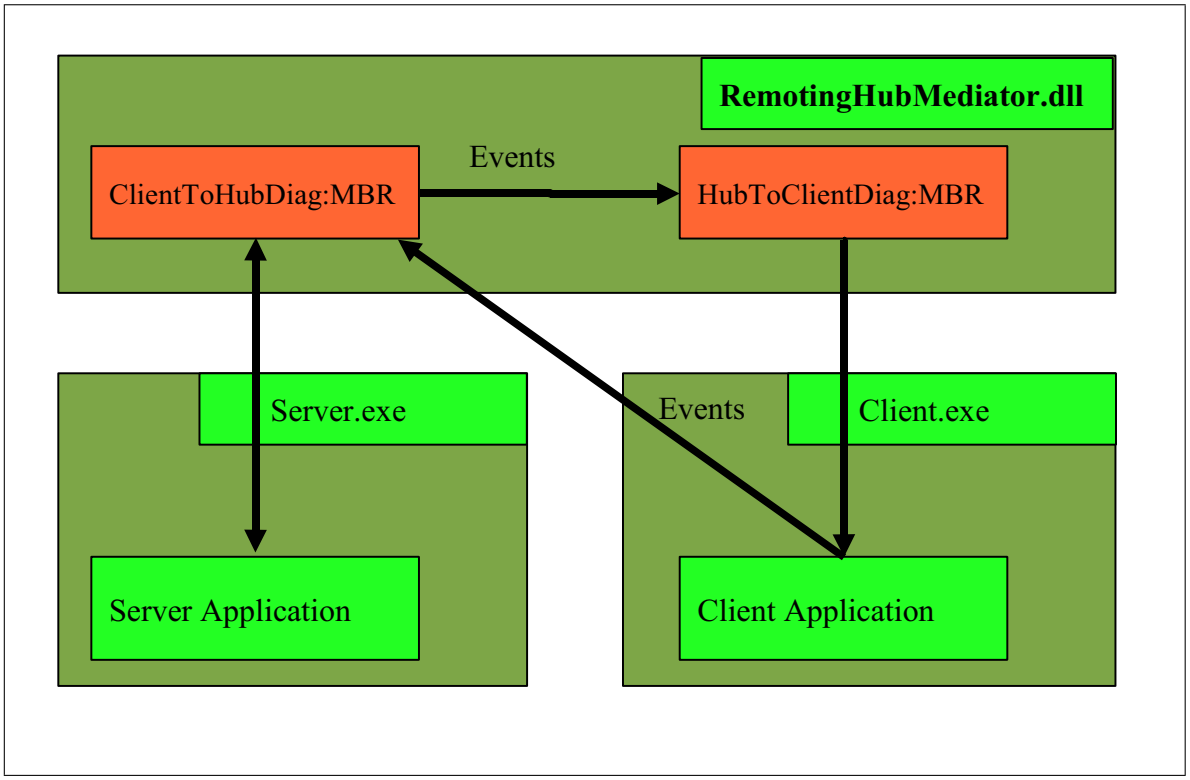

Abbildung 5.5.: Bidirektionale Kommunikation mit CallBack in RemotingHub

5.8. Nachrichtentransport

RemotingHub ist ein Server und kann deswegen mehrere Kunden oder Clients bedienen. Die Kunden eines Servers haben keinerlei Kenntnisse voneinander und stehen somit auch in keinem Bezug zueinander, außer der Tatsache, dass sie den gleichen Server verwenden. Somit stellt das RemotingHub einen zentralen Punkt dar, an den Anforderungen geschickt werden können. Außerdem steht in der Systemanforderung, dass die Clients eine asynchrone Kommunikationsbeziehung haben müssen und es soll in dem System eine direkte Kommunikation zwischen den SW-Komponenten vermieden werden. Man könnte sich fragen, wie genau dann die Clients zueinander in Beziehung stehen müssen, um diese Anforderungen zu erfüllen.
Eine entsprechende Lösung kann anhand des *Hub and Spoke* System (siehe Abschnitt 2.4.2) entwickelt werden. In einem *Hub and Spoke* Modell werden direkte Verbindungen nach Möglichkeit vermieden. Die Grundkonzept ist, dass der der Weg von einem Endknoten A zu einem Endknoten B dabei nicht direkt, sondern über einen Zentralknoten Z, der Hub führt. Die Verbindungen der Endknoten A und B zum Knoten Z bezeichnet man hierbei als Spokes (zu Deutsch „Speichen “). Wenn es einen Weg von einem Endknoten A zu einem Endknoten B im *Hub and Spoke* Modell mit einer direkten Kommunikation zwischen Clients C und D verglichen wird, soll nach diesem Konzept eine direkte Kommunikation auch in zwei Spoke-Kommunikation,

die über RemotingHub abläuft, unterteilt werden. Gemäß Abbildung 5.3 sollte eine Nachricht von EA nach RuleEngine in zwei Speichen übertragen werden. Die beiden Speichen sind:

1. EA nach Hub und
2. Hub nach RE

Logischerweise erfolgt der Rückweg bzw. der Weg der Rückantwort auf diese Nachricht genauso.

1. RE nach Hub und
2. Hub nach EA

Obwohl die *Hub and Spoke* Systeme mit dieser Eigenschaft einige Systemanforderungen erfüllen, bringt diese Eigenschaft auch ein weiteres Problem mit sich. Wie kann RemotingHub wissen, dass eine bestimmte Message von einem Client zum richtigen Client weitergeleitet wird, wenn der Absender nicht weiß bzw. das Ziel nicht angibt, wohin die Nachricht gesendet werden soll.
Zur Verdeutlichung dieser Frage wird das Publish-Subscribe-Modell von Abschnitt 2.4.3 eingesetzt. Das Grundkonzept dieses Modells ist die Veröffentlichung und Abonnierung der Events durch den Kommunikationspartner. Diese Kommunikationspartner sind in dieser Arbeit die SW-Komponenten des TraceMaintainer, also die Clients. Da in der TM-Anwendung jede SW-Komponente sich nur für bestimmte Nachrichten von anderen SW-Komponenten interessiert, ist die Veröffentlichung der Events eines Clients nur für den bestimmten, nicht aber für jeden Client interessant. Deswegen wird in dieser Arbeit die Veröffentlichung der Events nicht berücksichtigt, sondern das Hauptaugenmerk auf die Abonnierung der Events gerichtet. Es wird aus der Sicht des Systems davon ausgegangen, dass die Clients wissen, welche Events sie bekommen können und wie sie auf diese reagieren müssen. Der wichtigste Teil von diesem Modell ist der Broker, der die Verteilung der Nachrichten übernimmt. In Abbildung 5.6 ist das hier zu realisierende System, RemotingHub („Hub “ in Abbildung) zu sehen. EA1, EA2, RE1 und DB sind Clients. Die Clients können sowohl Publisher als auch Subscriber sein. Wobei spielt RemotingHub die Rolle des Brokers, der zur Nachrichtenverteilung dient.

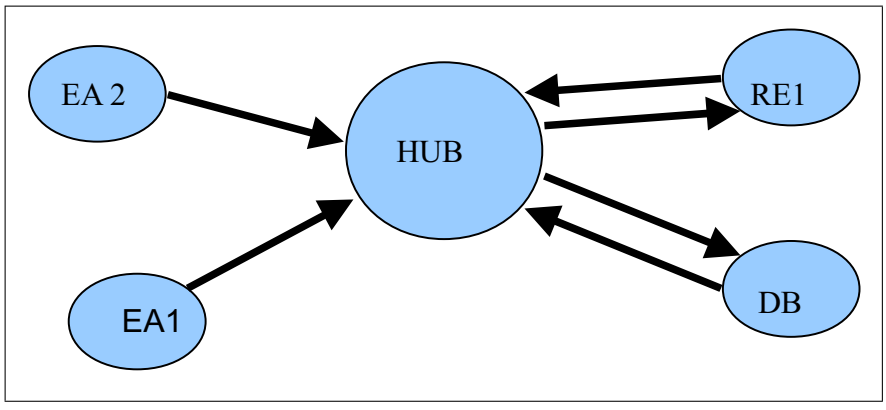

Abbildung 5.6.: Publish-Subscribe-Modell im RemotingHub

Zur Abonnierung des Events wird standardmäßig eine Konfigurationsdatei verwendet, in der steht, welcher Client welche Events immer bekommen soll. Diese wird dann bei der Client-Registrierung als initiale Abonnierung verwendet. Zudem haben die Clients auch die Möglichkeit, die Abonnierung während der Laufzeit durchzuführen. Somit haben die Clients mehr Flexibilität bei der Abonnierung ihrer Events, sodass sie auch nach ihrer Registrierung die fehlenden Abonnierungen vervollständigen können. Nach einer erfolgreichen Client-Registrierung kann jeder Client Nachrichten an RemotingHub verschicken. Da das System über die Client-Abonnierungen Bescheid weiß, kann es diese empfangenen Nachrichten entsprechend verteilen. Auf die Verteilung dieser Nachrichten wird im kommenden Kapitel detailliert eingegangen.

```
<?xml version="1.0" encoding="utf-8"?>
<TIMConfigurationScript>
  <OnJoiningClient ClientType="ToolAdapter" modelType="Enterprise
Architect">
    <OnFirstJoiningClient  model="ABC" host="XYZ" user="Burkut"
clientType="ToolAdapter" modelType"EA\glqq >
      <StartClient ClientType="CalculatorAPI" Model="?" modelType="?" >
</StartClient>
    </OnFirstJoiningClient>
  </OnJoiningClient>

  <OnJoiningClient>
    <OnFirstJoiningClient tim="?">
      <StartClient clientType="Logger" tim="?">   </StartClient>
    </OnFirstJoiningClient>
  </OnJoiningClient>

  <OnLeavingClient>
    <OnLastLeavingClient tim="?">
      <CloseClient clientType="Logger" tim="?">
      </CloseClient>
    </OnLastLeavingClient>
  </OnLeavingClient>

  <OnLeavingClient clientType="ToolAdapter" modelType="Enterprise
Architect">
    <OnLastLeavingClient model="?" user="?">
      <CloseClient clientType="CalculatorAPI" model="?" modelType="?">
      </CloseClient>
    </OnLastLeavingClient>
  </OnLeavingClient>
</TIMConfigurationScript>
```

Abbildung 5.7.: Session Verwaltung mittels XML

5.9. Umsetzung der Sessionverwaltung mittels XML

Nach der Anforderung muss es mit diesem Nachrichtentransportsystem möglich sein, die Sessions automatisch, ohne Einfluss von Entwickler bzw. von bestimmten Clients zu verwalten. Dafür muss das System in der Lage sein, bestimmte SW-Komponenten des TraceMaintainer zur Laufzeit zu starten und/oder zu stoppen. Warum muss es so gemacht werden? Der Hintergrund ist, dass RE zur Zeit eine PlugIn von EA und keine eigenständiges Modellierungstool ist. Ausserdem ist TS eine SW-Komponente, die Traceability-Link-Information verwaltet und gleichzeitig der RE und dem EA dient. Daraus kommt die Unterteilung in aktive und passive SW-Komponente. Das o.g. Starten und/bzw. Stopen der bestimmten SW-Komponenten bezieht sich auf die Passive, Weil jeder aktiven SW-Komponente (in dem Fall EA mit EG) eine RE und eine TS zugeordnet werden müssen, wenn man die Traceability-Link-Beziehungen in einem UML-Model anwenden möchte. Da die TM-Anwendungen bzw. SW-Komponenten dabei verteilt sind, kann eine SW-Komponente die Andere nicht zur laufzeit instanzieren bzw. anwenden. Deswegen muss der Server die Aufgabe übernehmen, jeder aktiven SW-Komponente eine RE und eine TS zuordnen. Damit der Server diese Aufgaben erledigen kann, müssen die Pfadangaben der RE- und TS-Anwendungen dem Server bekannt sein.

Zur Vermeidung von Missverständnissen in den folgenden Teilen der Arbeit müssen an dieser Stelle die Zusammenhänge zwischen An-und Abmeldung sowie Start und Stopp der Clients verstanden werden. In dieser Arbeit wurden deswegen die folgenden Vereinbarungen geschaffen und hier aufgelistet:

1. Die An- und Abmeldung von aktiven Clients verursacht das Starten und Stoppen von passiven Clients.
2. Server kann aktive Clients nicht starten, sie müssen selbst durch ihre Anmeldung dem Server bekannt machen.
3. Server kann nur passive Clients starten und stoppen. Danach findet ihre Anmeldung und Abmeldung statt.

Damit das System diese Anforderung erfüllen kann, wird eine Regelbasis erstellt. Somit kann dieses System automatisiert eigenständig arbeiten. Die technische Umsetzung erfolgt textbasiert in XML, um eine leichte Erweiterbarkeit und Anpassbarkeit zu ermöglichen. Dieser Lösungsansatz ist deswegen ausgewählt worden, weil diese Kriterien relativ flexibel und nicht fest definiert sein müssen. Die Rahmenstruktur der Regelbasis ist mittels XML umgesetzt worden. Dies ermöglicht eine sehr einfache Handhabung der Daten. Sie können in jedem Fall mit Hilfe eines Texteditors eingelesen und verändert werden. Somit hat man die Möglichkeit, die Regeln in Zukunft zu erweitern und anzupassen. Außerdem enthalten fast alle aktuellen Programmiersprachen generell umfangreiche Funktionen zur Verarbeitung von XML-Daten. Die Client-Anmeldung erfolgt logischerweise vem Client selbst. Die Abmeldung ist hingegen nicht immer. Sie kann gewöhnlich in den folgenden Fällen vorkommen:

- Abmeldung durch einen aktiven Clients selbst
- Der Client wird vom Nutzer angehalten
- Der Nutzer auf dem Client-Rechner meldet sich ohne Client-Abmeldung ab

- Der Client-Rechner ist ohne Client-Abmeldung heruntergefahren
- Es tritt einen Fehler beim Client auf und dieser ist dann nicht erreichbar, z.B.:
  - Stromausfall beim Client
  - Die Netzwerkprobleme

In Abbildung 5.7 ist die XML-Struktur der Regelbasis dargestellt. Alle durch Fragezeichen gekennzeichneten Stellen werden zur Laufzeit entsprechend vervollständigt und die anderen Stellen nur geprüft. Die zur Vervollständigung nötigen Informationen müssen bei der entsprechenden Aktion (Anmeldung bzw. Abmeldung) geliefert werden. Grundidee dieses Konzepts ist, eine Session automatisch zu bauen bzw. zu zerstören.
Jeder Client, der sich an- und abmeldet, liefert ein Reihe von Informationen, wie z.B:

- ClientType z.B. ToolAdapter, RuleEngine, traceStore
- User z.B. kAbula
- Model z.B. C:/myFiles/test.eap
- ModelType z.B. Enterprise Architect, DOORS, Matlab
- Host z.B. pi21, pi28

Die Informationen bzw. Clienteigenschaften werden im System nicht fest codiert, sondern variabel gehalten, da neue Informationen in Zukunft hinkommen können (z.B. toolVersion='EA7.5' oder os='Windows XP'). Im Moment gibt es nur sechs Sections, wie man in der XML-Struktur sieht.
OnJoiningClient und OnLeavingClient werden jedes Mal ausgeführt, wenn sich ein Client mit den angegebenen Eigenschaften anmeldet. OnFirstJoiningClient, OnLastLeavingClient werden ausgeführt, wenn sich der erste bzw. der letzte Client mit den angegebenen Bedingungen an- oder abmeldet. Wichtig ist hier, dass die Fragezeichen für die Werte des aktuellen, sich an- oder abmeldenden Clients stehen. StartClient und CloseClient sind selbsterklärend, die Parameter werden dem Client beim Starten übergeben und der Client meldet sich mit diesen Informationen am System an, oder aber die Informationen werden nur zur Überprüfung verwendet. Im erfolgreichen Fall wird dieser Client vom System abgemeldet und dessen Prozess gestoppt.

6. Implementierung

Ein weiterer Kernpunkt dieser Arbeit ist die Implementierung eines Prototyps, der das zu konzipierende Nachrichtentransportsystem realisiert. Dies Implementierung des Systems soll im Folgenden vorgestellt werden. Das Ziel der Implementierung ist, das System mit allen hier definierten Festlegungen vollständig lauffähig umzusetzen, um anschließend experimentelle Aussagen zur Leistungsfähigkeit und Zuverlässigkeit des Ansatzes treffen zu können.

6.1. Realisierung von RemotingHub

Zur Implementierung des Prototyps kam die Entwicklungsumgebung von Visual Studio von Microsoft mit der Programmiersprache C# im Framework .Net zum Einsatz. Die Entwicklungsumgebung steht den Studenten an der TU Ilmenau zum Zweck der Forschung frei zur Verfügung. Sie bietet den vollen notwendigen Funktionsumfang. Die Programmiersprache wurde deshalb ausgewählt, weil das System TraceMaintainer mit Hilfe von diesem .Net-Framework implementiert worden ist. Dazu kommt noch die Anforderung des Betreuers, dass er bei der Implementierung des Prototyps aus Kompatibilitätsgründen ein möglichst gleiches Framework wie TraceMaintainer verwenden möchte, damit die zukünftige Weiterentwicklung des Prototyps für das Forschungsteam einfacher wird.
Ziel des zu entwickelnden Systems soll sein, eine möglichst schnelle Kommunikation von SW-Komponenten der TraceMaintainer-Anwendung lokal auf einem Rechner und Rechnergrenzen hinweg zu ermöglichen. Eine Grundlage hierfür bildet die Kommunikationsinfrastruktur, die im letzten Kapitel konzipiert wurde. Dabei spielt das System die Rolle eines softwaremäßigen Hubs, der die Nachrichten von einem Source-Client zu einem anderem Ziel-Client korrekt übermittelt. Dabei sollte dem System keine Hilfe vom Nutzer nötig sein, sondern alle Aufgaben sollen vom System allein vollständig automatisiert erledigt werden. Im Folgenden wird das System noch etwas ausführlicher beschrieben.
Damit das Nachrichtentransportsystem eine Verarbeitung der eingehenden Nachrichten gemäß im Abschnitt 5.3 definierter Anforderungen durchführen kann, müssen die folgenden grundlegenden Voraussetzungen geschaffen sein.

6.1.1. Systemvoraussetzungen

Zum Einen müssen die Verbindungen zwischen Clients und Server zum Zeitpunkt ihrer Kommunikation hergestellt sein. Dies ist überhaupt die Grundvoraussetzung. Zum Anderen müssen die Clients für gewünschte Events von gewünschten Clients korrekt abonniert sein. Wie korrekt abonniert werden kann, wird im Abschnitt diskutiert. Des Weiteren wird vorausgesetzt,

dass die Clients immer die nötigen Informationen (Modell, Host, Clienttype, usw.) liefern, um einen bestimmten Client an- und abmelden sowie die anderen Systemaufgaben erledigen zu können. Damit das System auf einem Rechner einwandfrei läuft, wird vorausgesetzt, dass der Systemanwender .Net-Version 3.5 verwendet.

6.1.2. Systemkomponenten

Ausgehend von den konzeptionellen Betrachtungen vom letzten Kapitel lassen sich klar definierte Anwendungsfälle (siehe Abbildungen 6.2 und 6.1) für das System erarbeiten, die das Projekt umsetzen muss. Beliebig viele Anwender sollen einen solchen einfach zu benutzenden Nachrichtentransport derart anwenden können, dass eine Nachricht, die Client sendet, an bestimmte angeschlossene Clients übermittelt wird. Folgende Use-Cases soll das Nachrichtentransportsystem, RemotingHub unterstützen (vlg. Abbildung 6.1):

- **Registerierungscheck**: Ein Client registriert sich bei RemotingHub mit bestimmten Eigenschaften. Diese Informationen sollen nach vorgegebenen Regeln teilweise überprüft und teilweise vervollständigt werden.
- **Sessionverwaltung**: Wenn die Registrierung erfolgreich ab lief, dann wird eine Session neu erstellt oder die vorhandene Session weiter verwaltet.
- **Starten passiver Clients**: Bei erfolgreicher Überprüfung der Clients (in der Regel nur aktive Clients) werden die passiven Clients sowie RE, TS und Logger gestartet.
- **Send Message**: Clients und der Server können Messages zueinander senden, wobei der Server eine Message noch weiterleiten kann.
- **ClientControl**: Jeder Client, der den Registerierungscheck bestanden hat, wird in der Client-Liste auf dem Server aufgenommen. Anhand der Liste werden die Nachrichten weitergeleitet. Andernfalls können sie auch aus der Liste gelöscht werden.
- **Stop Client**: Ein Client meldet sich vom Server ab bzw. der Server lässt den Client abmelden. In beiden Fällen erhält der abgemeldete Client keine Informationen mehr zugestellt.

Der Server (RemotingHubService) übernimmt im Wesentlichen folgende Aufgaben: Zuverlässige Nachrichtenübermittlung und -verwaltung, Client-An- und Abmeldung, Sessionaufbau und -verwaltung. Zur Herstellung der Kommunikation wird der Vermittler, der RemotingHubMediator benötigt und danach können die Nachrichten weitergeleitet werden. Somit ist klar, dass sich RemotingHub strukturell aus diesen beiden Komponenten zusammensetzt, wobei der RemotingHubMediator für den Server (RemotingHubService) und alle Clients eine gemeinsame Klassenbibliothek ist. Das ist praktisch hier problemspezifisch konzipierte Middleware, mit dessen Hilfe die ganze Kommunikation zwischen Server und Clients möglich gemacht wird.
Folgende Use-Cases sollen das Systemteil RemotingHubMediator wesentlich unterstützen (vlg. Abbildung 6.2):

- **Create Connection**: Nachdem der Server den Typ *Mediator* als bekannten Typ registeriert hat, kann der Client ein Proxyobjekt von diesem Typ auf dem Client erstellen, womit dann eine logische Verbindung existiert. Dann können die Clients sich mit ihren Eigenschaften beim Server anmelden.

- **Send Message to Server**: Wenn ein Client dem Server eine Message sendet, wird sie einfach weitergeleitet.
- **Send Message To Client**: Wenn der Server eine Nachricht an einen Client sendet, wird sie genauso weitergeleitet. Dies erfolgt jedoch nur, wenn der Client sich als *Callback*-Object dem Server bekannt gemacht hat.

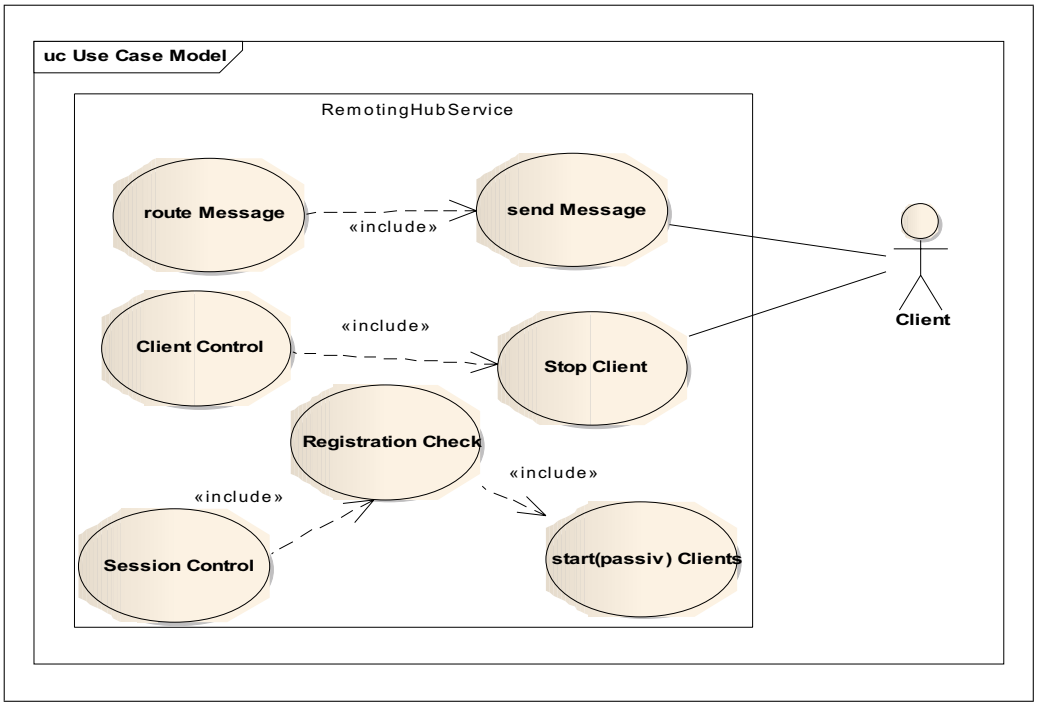

Abbildung 6.1.: Use Case Modell des Servers

Ein paar Einschränkungen grenzen die Funktionalitäten des Systems ein, um das System möglichst klar und effizient zu gestalten:

- Nur diejenigen Clients, die in RemotingHub angemeldet sind, erhalten die Nachrichten.
- Die Anwendung muss eigentlich im Internet funktionieren. Sicherheitstechnische Aspekte für eine Anwendung im Internet sind jedoch nicht vorgesehen.
- Wenn ein Client sich abmeldet, verschwindet er vollständig aus dem System und es bleibt keine Information im System erhalten.
- Es wird im System mindestens eine Client-Gruppe geben, die aus angemeldeten Clients bestehen. In diesem Fall wird in dieser Arbeit von einer Session gesprochen.

Wie Abbildung 6.3 zeigt, besteht der RemotingHubMediator aus zwei Teilen - ClientConnector und Mediator. Der Mediator ist wie der Name bereits besagt, ein Vermittler zwischen den Clients und dem Server. Ohne Beeinflussung des Inhalts werden sie entweder in Richtung Server oder Clients weitergeleitet.

Im Gegensatz dazu ist der *ClientConnector* eine abstrakte Klasse, welche jede Klasse, die eine Kommunikation zum Server herstellen möchte, implementieren muss. Bei der Implementierung der Methode *StartClient()* muss der Client darauf achten, dass er vor der Registrierung unbedingt ein remotefähiges *Callback*-Objekt erzeugen und an den Server übergeben muss, damit er später wieder Nachrichten vom Server bekommen kann.

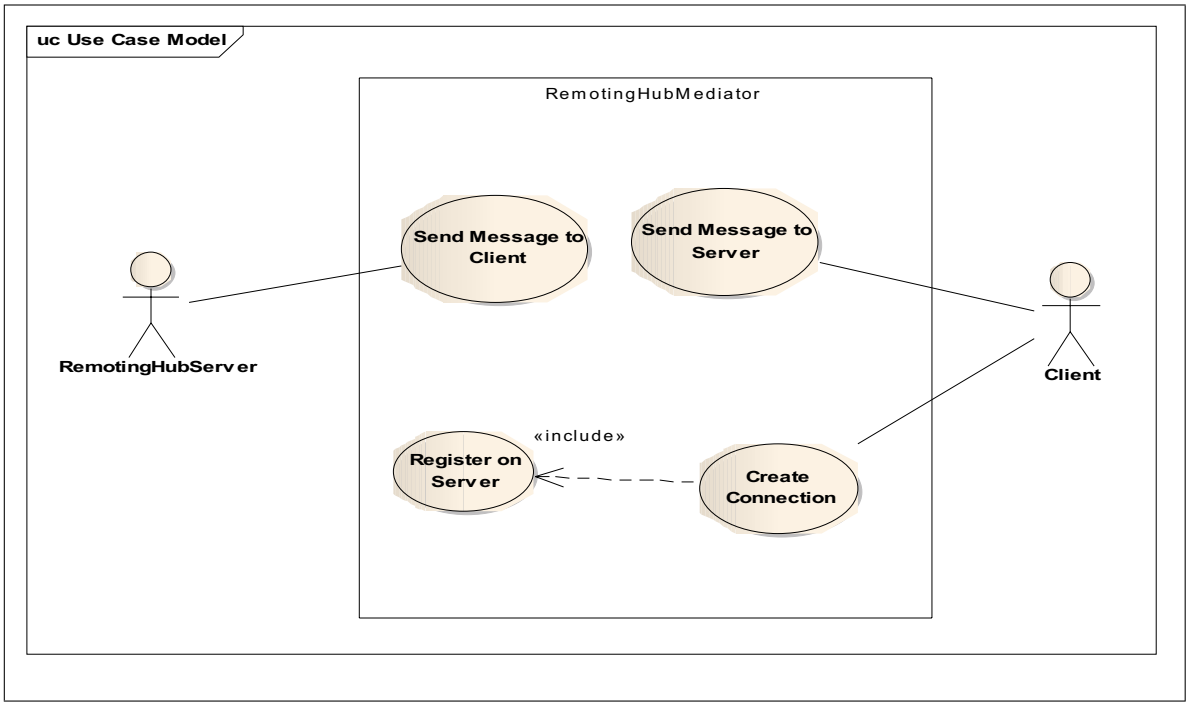

Abbildung 6.2.: Use-Case-Modell des RemotingHubMediator

6.2. Realisierung mit .NET Remoting

.NET Remoting beruht, ähnlich wie RPC (siehe Abschnitt 3.1), auf der Veröffentlichung von Methoden einer Klasse. Der Methodenaufruf kann dadurch anderen, auch den entfernten Clients, zur Verfügung gestellt werden. Um die Use-Cases in Abbildung 6.2 zu realisieren, wurde zunächst eine lokale Klasse erstellt, die verschiedene Methoden für die Kommunikation mit dem Server, RemotingHubServices bereitstellt. Die Programmierung erfolgte in der Sprache C# mit Hilfe des Visual Studio 2008. Der Rumpf der Klasse sieht so aus:

```
public class Mediator
{ ... }
```

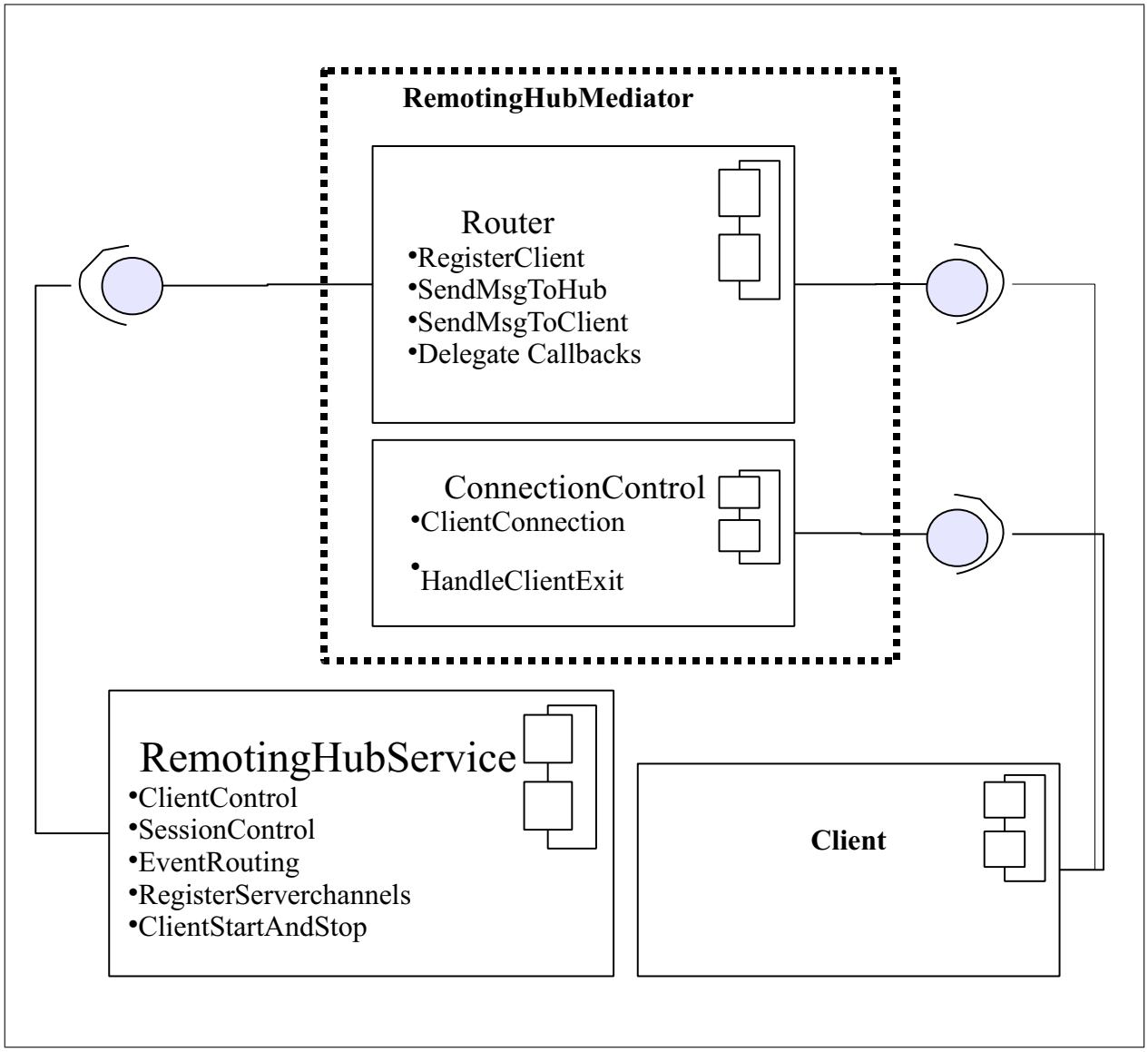

Abbildung 6.3.: Beziehungen von RemotingHubMediator, RemotingHubService, Client

Die durch diese Klasse (Einzelheiten können in Anhang A.2 eingesehen werden) bereitgestellten öffentlichen Methoden ermöglichen es, die Clients anzumelden und die Nachrichten sowohl an Server als auch Clients zuzusenden. Obwohl sich der Quellcode dieser Klasse, *Mediator* als DLL kompilieren lässt, kann in diesem Zustand aber beispielsweise nur von einem Windowsprogramm auf demselben Rechner instantiiert werden. Um sie auch anderen Prozessbereichen oder Rechnern zugänglich zu machen, bietet das .NET-Framework die Möglichkeit, die Klasse remotefähig zu machen. Dazu wird Mediator vom *MarshalByRefObject* abgeleitet:

```
public class Mediator : MarshalByRefObject
{ ... }
```

Dies allein ist aber noch nicht ausreichend. Der Serverprozess muss für dieses Objekt noch einen URI (Uniform Resource Identifier) bereitstellen. Dafür kann man beispielsweise IIS (Internet Information Service) verwenden. Dies kann jedoch nachteilig sein, sodass man für den Kommunikationskanal nur das http-Protokoll einsetzen muss. Um aber den Vorteil von .NET Remoting, nämlich die Verwendung des schlankeren TCP-Protokolls, ausnutzen zu können, wird diese Variante nicht verwendet. Um einen noch besseren Effekt bei der Interprozesskommunikation zu bekommen, wurde das IPC-Protokoll für den Kommunikationskanal zwischen den Clients und dem Server, die sich auf einem Host befinden, verwendet. Dementsprechend wurde ein Programm geschrieben, welches die Klasse serverseitig aktiviert:

```
using System;
using System.Runtime.Remoting;
using System.Runtime.Remoting.Channels;
using System.Runtime.Remoting.Channels.Tcp;
using System.Runtime.Remoting.Channels.Ipc;
class Server
{


    private static void RegisterTCPChannel()
    {
      Hashtable h = new Hashtable();
      h.Add(ïmpersonationLevel", "None");
      h.Add(äuthorizedGroup", Account.Value);
      h.Add("name", SServerChannel");
      h.Add("port", 9001);
      // Formatierung der Sinkprovider
      BinaryServerFormatterSinkProvider provider =
                    new BinaryServerFormatterSinkProvider();
      provider.TypeFilterLevel =
            System.Runtime.Serialization.Formatters.TypeFilterLevel.Full;
      TcpChannel serverChannell = new TcpChannel(h, null, provider);
      ChannelServices.RegisterChannel(serverChannell, false);
      // register a WKO type in Singleton mode
      WellKnownObjectMode mode = WellKnownObjectMode.Singleton;
      WellKnownServiceTypeEntry entry= new WellKnownServiceTypeEntry
```

```
    (typeof(Mediator),"RemotingHub", mode);
      }


  private void RegisterIPCChannel()
  {
    IDictionary prop = new Hashtable();
    prop["portName"] = "localhostIPCChannel";
    prop["name"] = ÏPCServerChannel";      //Ensures Unique name.
          // Formatierung der Sinkprovider
    IpcChannel serverChannel = new IpcChannel(prop,
                clientProv, serverProv);
    ChannelServices.RegisterChannel(serverChannel, true);
    RemotingConfiguration.RegisterWellKnownServiceType(typeof(Mediator),
          "RemotingHub", WellKnownObjectMode.Singleton);
  }


}
```

Zunächst legt das Serverprogramm einen Kanal an. Dieser stellt eine Leitung zwischen dem remotefähigen Objekt und den Clients her. Der angegebene Port ist frei wählbar und in einer Firewall evtl. freizugeben. Anschließend wird der Kanal registriert. Der dritte Schritt ist die Registrierung der Klasse, *Mediator* unter dem URI RemotingHub. Der dritte Parameter für die Registrierung der Klasse gibt an, ob für jeden Aufruf der Klasse eine neue Instanz angelegt werden soll (*SingleCall*) oder ob nur eine Instanz der Klasse für alle Aufrufe angelegt wird (*Singleton*). Solange das Serverprogramm läuft, können Instanzen der Klasse auf entfernten Clients erzeugt und ihre Methoden aufgerufen werden.
Der Quellcode eines Client-Programms muss, um nun auf die remotefähige Klasse zugreifen zu können, einfach die statische Methode ConnectClient von der Klasse, *ClientConnection*, aufrufen. Der Rest wird dann automatisch erledigt.

```
public static object connectClient(string serverIP, int port)
{
  string channel = ConfigureClient(serverIP);
  if (channel == „ipc„)
  return getIPCChannelProxy();
  else
  return getTCPChannelProxy(serverIP,port);
}
```

Dafür muss der Client jedoch die Server-IP und die Port-Nummer angeben. Zunächst wird wiederum ein Kanal auf dem Client angelegt und registriert. Das passiert in *ConfigureClient (serverIP)*. Es ist standardmäßig so eingerichtet, dass anhand der übergebenen Server-IP entschieden wird, ob der aktuelle Client am besten ein TCP- oder IPC- Protokoll für seine Kommunikation benötigt. Aber man kann auch über die Config-File einstellen, dass man auch für die lokale

Kommunikation (Interprozesskommunikation) ein TCP-Protokoll benötigt. Anschließend wird die remotefähige Klasse registriert und ein Objekt auf folgende Art angelegt:

```
private static object getIPCChannelProxy()
{
  object obj = RemotingServices.Connect(typeof(RemotingHubMediator.Mediator),
  ïpc://localhostIPCChannel/RemotingHub");
  return obj;
}
private static object getTCPChannelProxy(string serverIP, string port)
{
  object obj = RemotingServices.Connect(typeof(RemotingHubMediator.Mediator),
  "Tcp://"+ serverIP + ":"+port+"/RemotingHub");
  return obj;
}
```

Wichtig ist aber, dass innerhalb des aktuellen Client Ordner (wo der Client ausgeführt wird) der kompilierten Klasse, ein *Mediator* bzw. *RemotingHubMediator* vorhanden sein muss. Ohne diese Vorlage kann kein Proxy auf dem Client angelegt werden. Im Deployment-Diagramm auf Abbildung 7.3 wird diese Beziehung gezeigt, welche Dateien und Programme notwendig sind, um RemotingHub mittels .NET Remoting von einem entfernten Client aus anzusprechen.
Die im kommenden Kapitel beschriebene Testumgebung stellt die Kommunikationsbeziehungen der verteilten TM-Anwendung dar, die auf die Klasse, Mediator, mittels .NET Remoting zugreift. Mit dieser Testumgebung wurde demonstriert, wie man diese Technologie für die Kommunikation der SW-Komponenten der verteilten TraceMaintainer (TM)-Anwendung über das Internet einsetzen könnte.

6.3. Kommunikationsablauf

In Abbildung 6.3 sieht man, wie die Komponenten von RemotingHub zu den Clients in Beziehung stehen. Es ist nicht schwer zu erkennen, dass dabei die Clients und Server die gleiche Komponente *RemotingHubMediator* auf unterschiedliche Art nutzen. Während der Server den *RemotingHubMediator* zum Registrieren eines Channels (dabei wird der Typ des gewünschten Remoteobjekts nötig) nutzt, braucht ein Client ihn zur Erzeugung eines Proxy-Objekts, das die Kommunikation zum Server ermöglicht. In diesem Abschnitt wird ein Überblick über den Kommunikationsablauf zwischen Clients gegeben. Es wurde einen Ansatz zur An- und Abmeldung sowie Start und Stop eines Clients im Abschnitt vorgestellt. Anhand dieses Ansatzes werden im Folgenden die Abläufe für die Kommunikation erläutert. In Anhang B.1 befindet sich auch ein Sequenzdiagramm, welches einen knappen Überblick über den ganzen Ablauf der Kommunikation zwischen dem Server und den Clients darstellt.
Der Kommunikationsaufbau:Unter dem Kommunikationsaufbau wird in dieser Arbeit die Sende- und Empfangsbereitschaft von Nachrichten zwischen Server und Client verstanden. Mittels RemotingHub kann sich jeder Client mit Hilfe der Schnittstelle *ClientConnector* zu dem

Server verbinden. Dabei wird nur ein Proxy-Objekt von einem remotefähigen Typ *Mediator* erzeugt. Dieser Typ muss zunächst, wie im letzten Abschnitt erläutert, auf der Serverseite als bekannter Typ registriert sein. Wenn der Client über Informationen wie Objekttyp, URL, usw. verfügt, kann sich der o.g. Proxy von dem Remoteobjekt erzeugen und nutzt dieses zum Zugriff auf das Objekt. In diesem Fall spricht man in .Net Remoting von *Marschall by Reference*, da der Proxy eigentlich eine Referenz von dem Remoteobjekt auf dem Server ist. Es gibt noch eine andere Variante, *Marschall by Value*, wobei das Proxyobjekt keine Referenz, sondern eine Kopie vom Remoteobjekt auf dem Server ist. Aber dies kam im Rahmen dieser Arbeit nicht zur Anwendung. Nach einer erfolgreichen Erstellung eines Proxy-Objekts kann der Client mit der Anmeldung beginnen. Eine Anmeldung kann theoretisch von jedem Client gemacht werden. Aber es wird in diesem Zusammenhang von einer Anmeldung eines aktiven und passiven Clients getrennt betrachtet, weil passive Clients sich anhand der Informationen von aktiven Clients anmelden und diese Anmeldung i.d.R. von den passiven Clients nicht „aktiv “ ausgelöst wird. D.h. Der Server starter sie und lässt sie anmelden. Wie schon im Abschnitt erwähnt, sollten die Clients sich für die Events von anderen Clients abonnieren (siehe Abschnitt). Standardmäßig erfolgt die Abonnierung so, wie es in der Konfigurationsdatei gegeben ist. Zur Anmeldung beim Server ruft man jetzt die Registrierungsmethode bei Mediator *RegisterClient()* auf.

Nachrichten auf dem Server: Das ist nächste Phase des Kommunikationsablaufs. In dieser Phase werden die empfangenen Nachrichten je nach ihren Typ verarbeitet. Wenn ein Client sich an- und abmelden möchte, werden dessen Daten in die Client-Liste des Servers eingetragen bzw. daraus gelöscht sowie eine Rückmeldung gesendet. Ansonsten werden die Nachrichten an die anderen Komponenten bzw. Clients weitergeleitet. Darauf wird im nächsten Abschnitt näher eingegangen.

Der Kommunikationsabbau: Unter Kommunikationsabbau versteht man, dass eine vorhandene Kommunikation abgeschlossen wird. In RemotingHub spricht man von einem Kommunikationsabbau, wenn ein Client bzw. Clientprozess sich abmeldet oder gestoppt wird. Ein Kommunikationsabbau ist aus verschiedenen Gründen möglich ebenso wie ein Systemfehler, eine aktive Anforderung zur Session-Zerstörung von einem Client, etc. Da in RemotingHub aktive und passive Clients gleichzeitig existieren können, muss man zur ihrer Abmeldung auf zwei unterschiedliche Weisen vorgehen. Während die passiven Clientprozesse vom Server aus gestoppt werden, werden die aktiven Clients nur per Message benachrichtigt, sodass diese dann abgemeldet bzw. gestoppt werden müssen. Die aktiven Clients müssen darauf vorbereitet sein und auf bestimmte Weise reagieren. Zu beachten ist dabei, dass die An- und Abmeldung des Clients nach der im Abschnitt beschriebenen Regelbasis erfolgt. Der Server kann also die passiven Clients starten und stoppen, nicht aber die aktiven Clients. Um die aktiven Clients zu stoppen, schickt der Server nur eine Stop-Message an die Clients. Anhand dieser Information können diese dann entweder durch Zerstörung des Proxyobjektes die Verbindung zu Server abbrechen oder aber die Anwendungen beenden. Natürlich löscht der Server entsprechend die gespeicherte Client-Information vom Server aus.

6.4. Funktionalitäten des Servers

Wenn Nachrichten beim Server ankommen, muss der Server sie entsprechend verarbeiten. Unter Verarbeitung wird hier nicht etwa eine Veränderung des Inhalts verstanden, sondern lediglich die Registrierung eines Clients auf dem Server oder die Weiterleitung einer bestimmten Nachricht. Zur Erledigung dieser beiden wichtigen Aufgaben sollte der Server noch weitere Fähigkeiten besitzen. In diesem Abschnitt werden diese Server-Funktionalitäten vorgestellt.

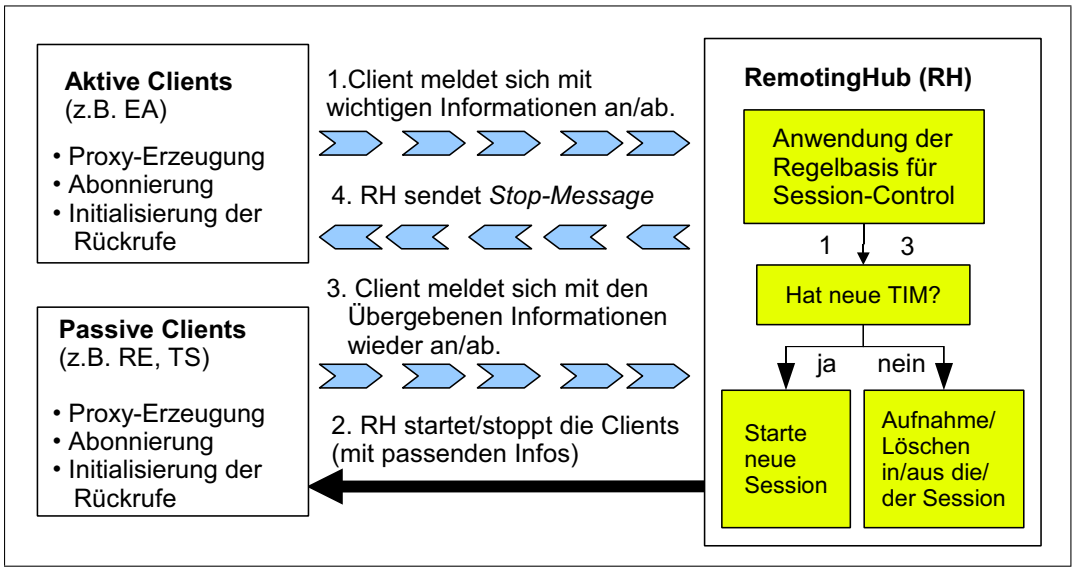

Abbildung 6.4.: Session-Verwaltung im RemotingHub

6.4.1. An- und Abmeldung eines Clients

Wenn eine Nachricht zur Anmeldung an den Server geschickt werden soll, wird sie über *RegisterClient()* im Mediator übermittelt. Der Server verarbeitet die Client-Anmeldung auf zuvor definierter Regelbasis. Wenn die gelieferten Informationen zu den jeweiligen Regeln passen, dann werden sie registriert und ihre Informationen in einer Hash-Tabelle gespeichert. Bei ihrer Abmeldung werden diese gespeicherten Daten wiederum nach Abmelde-Regeln geprüft. Wenn sie diese Überprüfung bestehen, dann werden sie abgemeldet. Es geht hier nicht bloß um ihre Anmeldung, sondern um den Start und Stop weiterer Clients bzw. Zerstörung einer vorhandenen Session.

6.4.2. Sessionverwaltung

In RemotingHub sind Sessions ein abstraktes System, und die Verwaltung der Sessions ist eine der wichtigsten Systemaufgaben. Unter einer Session wird in dieser Arbeit eine Gruppe

von Clients verstanden, die eine oder mehrere relevante Modelle zusammen verarbeiten, die Traceability-Link-Beziehungen zueinander haben. Angenommen, zwei Entwickler verarbeiten ein Klassendiagramm in einer räumlich verteilten Netzwerkumgebung (angenommen, dass die beiden Entwickler auf einen und denselben Fileserver Zugriff haben). Dann melden sich die beiden Entwickler als aktive Clients beim Server an. Zusätzlich muss noch RE, TS und ein Logger zur Protokollierung gestartet werden. Dann bilden diese fünf Clients eine Session. Jetzt ist die Frage, wie die Session-Zugehörigkeit bestimmt wird. Zur Verdeutlichung dieser Frage wurde das Konzept TIM (Traceability-Information-Model) eingeführt.
TIM ist eine Art von Session ID, mit der identifiziert werden kann, in welcher Session eine bestimmte Nachricht verteilt bzw. einfach weitergeleitet werden kann. Das Konzept ermöglicht es herauszufinden, welches Modell welche TIM hat. Wenn ein Client den Modellnamen liefert, wird dann nach der Anmelde-Regel geprüft und entschieden, ob eine weitere Session nötig ist. Wenn das Modell ein TIM hat, das gerade benutzt wird, wird der Client in die vorhandene Session eingefügt. Ansonsten wird eine weitere Session mit entsprechenden Clients geöffnet. Natürlich können die Clients aus einer Session zurücktreten bzw. ein Client wird aus der Session gedrängt, wenn er eine bestimmte Zeit lang nicht erreichbar ist bzw. er sich aktiv abmeldet. Die Abbildung 6.4 zeigt, wie die Sessions in RemotingHub auf- und abgebaut werden. In jedem Fall (An- und/oder Abmeldung) wird die Regelbasis angewendet (siehe Schritt 1 und 3), um das entsprechende TIM zu bekommen, danach wird entschieden:

- ob der Client eine neue Session benötigt. Wenn das der Fall ist, werden weitere Clients gestartet (siehe Schritt 4).
- ob der Client in die vorhandene Sesseion aufgenommen wird.
- ob der Client einfach aus der Session gedrängt wird. In diesem Fall wird wie folgt vorgegangen:
  - Aktive Clients werden per *Stop-Message* benachrichtigt (siehe Schritt 4).
  - Passive Clients (Clientprozesse) werden einfach gestoppt (siehe Schritt 2).

6.4.3. Routing der Nachrichten

Die Kernaufgabe von RemotingHub ist es, Nachrichten von Knoten A nach Knoten B weiterzuleiten. Damit das System diese Aufgabe erledigen kann, sind zunächst Client-Abonnierungen für bestimmte Events bzw. Event-Quellen erforderlich. Die Abonnierungen müssen entweder bei der Client-Anmeldung oder zur Laufzeit mindestens einmal gemacht werden, ansonsten bekommt dieser Client keine Nachrichten von Server erteilt. Ein Client kann mehrfach abonnieren, sodass er mehrere unterschiedliche Abonnierungsregeln bzw. -kriterien an den Server übergibt. Die Abonnierung erfolgt nach einem erweiterten DNF-Prinzip (Disjunctive Normal Form) in boolescher Algebra [WH06]. Eine Abonnierung hat eine gültige Abonnierungsform, wenn sie eine Disjunktion von Abonnierungsregeln ist. Eine Abonnierungsregel wird ausschließlich durch die konjunktive Verknüpfung von Literalen gebildet. Eine Literale ist ein Ausdruck, der aus einer Variablen und deren Wert besteht und durch Gleichheitszeichen „=“ getrennt ist. z.B „Events = ChangeEvent “ , „host = burkut “ sind gültige Literale in der Abonnierungsregel. Im Unterschied zu einer Literalen in DNF können die Literalen hier mehrwertig sein. So ist z.B.

erlaubt „Events= ChangeEvent;Createlink;GetLink;“ . Die weiteren Vereinbarungen sind hier aufgelistet:

1. Disjunktion der Abonnierungsregel wird durch „$“ ausgedrückt, z.B. Regel1$Regel2.
2. Die Konjunktion in der Abonnierungsregel wird durch „§“ ausgedrückt: Events= ChangeEvent§host=burkut§
3. Da in der Abonnierungsregel eine Variable mehrwertig sein kann, müssen diese Werte durch Semikolon „;“ wie z.B. in der Art von „Events= ChangeEvent;Createlink;GetLink;“ getrennt werden.

Folgendes sind die Beispiele für gültige Abonnierungen:

1. Events=ChangeEvent;Createlink;GetLink;$User=Martin§host=burkut
2. host=burkut§TIM=XYZ

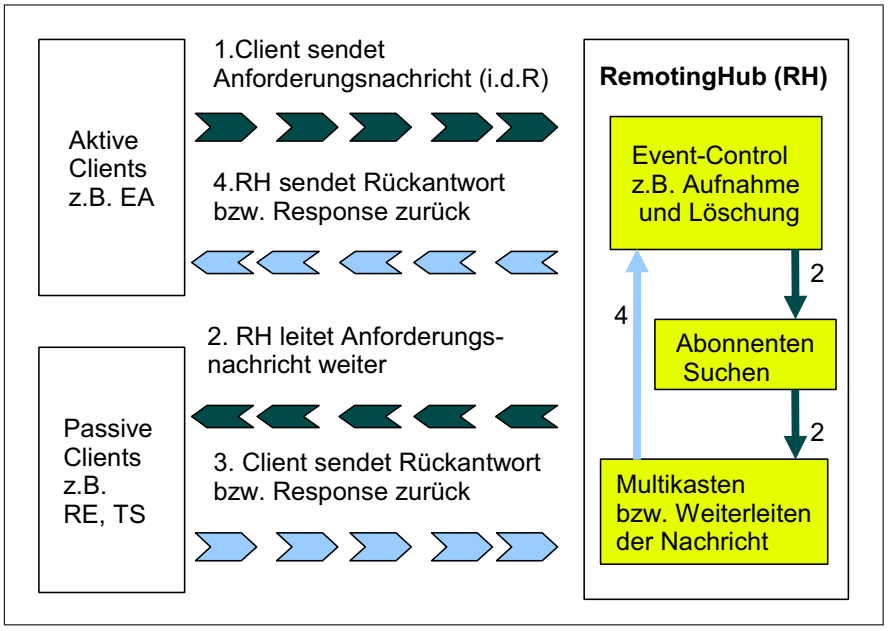

Abbildung 6.5.: Einen Blick in den RemotingHub

Die Abonnierung im ersten Beispiel besteht aus 2 Regeln. Mit der ersten Regel bekommt ein Client alle *ChangeEvent*s, *Createlink-Event*s und *GetLink-Event*s. Mit der zweiten Regel abonniert ein Client für die Events, und zwar die von dem User *Martin* und vom Host *burkut*. Das zweiten Bespiel sagt aus, dass ein Client für alle Events vom Host Burkut abnoniert hat.
Damit RemotingHub die eingehenden Messages systematisch verwalten kann, wurden Event-Typen wie Response, Request, Stop, RegisterClient und Exception ins System eingeführt. An dieser Stelle wird angenommen, dass die Clients auch den Typ ihrer Nachrichten immer angeben, sonst kann das System sich möglicherweise nicht wie gewünscht verhalten. Darunter sind zur Weiterleitung einer Nachricht nur Response und Request wichtig. Wenn eine Nachricht

beim Server ankommt, wird sie anhand dieser o.g. Event-Typen erst einmal sortiert. Danach wird nach den Abonnenten für dieses Event gesucht.
Dabei wird zunächst der Absender der Nachricht festgestellt. Vom Absender sind die Informationen interessant, die zu seiner Anmeldung nötig waren. Diese Informationen werden im Folgenden einfach Anmeldeinformation genannt. Außerdem sollte jede Nachricht einen gültigen Event-Namen wie *ChangeEvent* oder *GetLink*, die in der Regel auch bei Clients bekannt sind, verwenden. Zur Auswahl der Clients wird die aktuelle Client-Liste durchlaufen. Beim Durchlauf werden nur die Abonnierungsregeln des jeweiligen Client herausgenommen und jeweils mit den *Anmeldeinformationen* des Absenders sowie dem Event-Namen der aktuellen Nachricht abgeglichen. Wenn es nur einen Treffer gibt, dann wird dieser Client in die Ziel-Client-Liste aufgenommen. Wenn ein Client mehrere Abonnierungsregeln hat, wird die aktuelle Nachricht nur dann an den Client gesendet, wenn eine von allen Abonnierungsregeln erfüllt ist.

Die Abbildung 6.5 zeigt, wie die eingehenden Nachrichten in RemotingHub verteilt wird. Hier wird nur von den Event-Typen wie Response und Request gesprochen. Wenn eine Anforderungsnachricht in RemotingHub ankommt(Schritt 1 im Bild), wird sie entsprechend in die Event-Control (bzw. Nachrichtenliste) aufgenommen und dann wird nach den Abonennten der Nachricht gesucht. Wenn es Abonnenten gibt, die sich für die Nachricht interesieren, wird diese Nachricht im Schritt 2 an diese Abonnenten weitergeleitet. Nachdem die Abonnenten die Nachricht bekommen haben, werden sie die Nachricht entsprechend verarbeiten und erstellen Rückantworten an den ursprünglichen Absender der Nachricht. Im Schritt 3 wird die Rückantwort wieder an den Server gesendet. Da die Rückantworten an eindeutige Ziele geschickt werden, braucht der Server dieses Mal nicht nach dem Abonenten suchen, sondern kann direkt weiterleiten. Welche Rolle genau EventControl in RemotingHub hat, wird im Abschnitt 6.6 noch näher beschrieben.

6.5. Fehlersemantik

Das Ziel von RPC ist, die Kommunikation zu verbergen, in dem dafür gesorgt wird, dass entfernte Prozeduraufrufe genau wie lokale Prozedureaufrufe aussehen. Solange sowohl Clients als auch Server perfekt funktionieren, erledigt RPC seine Aufgabe, ausgezeicht. Ein Problem kann erst auftreten, wenn Fehler ins Spiel kommen [TS03]. Da die Interaktion zwischen den Clients der verteilten TraceMaintainer-Anwendung mit Hilfe der zugrunde liegenden Netzwerkkommunikation implementiert ist, vergrößert dies die Anzahl der Kommunikations- bzw. Interaktionsfehler. Hier kann der Unterschied zwischen lokalen und entfernten Aufrufen nicht immer ganz einfach maskiert werden. Diese können sein:

1. Die Anforderung geht verloren oder hat eine Verzögerung.
2. Die Rückantwort geht verloren oder erfährt eine Verzögerung.
3. RemotingHub oder Client können zwischenzeitlich abgestürzt und dadurch nicht erreichbar sein.
4. Aufgrund von zwischenzeitlichen Netzwerkausfällen können Client und Server nicht erreichbar sein.

Um auf solche Fehler und Ausfälle zu reagieren, wurde schon eine Ausnahme-Behandelung ins System eingeführt. Fällt der Server aus, nach dem die Anforderung den Server erreicht hat, so gibt es keine Möglichkeit für den Client, dies herauszufinden. Fällt der Client aus, während er eine Anfrage angestoßen hat und auf die Rückantwort wartet, so ist für den Server kein Partner mehr vorhanden, der ihm das Abfrageergebnis abnimmt. Sollten sowohl Server als auch Client ausfallen, wird für den Erhalt der Nachricht keine Garantie gegeben.
RemotingHub ist jedoch in der Lage, auf diese Ausfälle zu reagieren. Wenn der Server mitbekommt, dass der Client ausfällt bzw. sich selbst zerstört, löscht der Server alle Informationen über diesen Client und aktualisiert sofort die Client-Liste, damit eine Nachricht nicht erneut an diesen Client geschickt wird. Wenn der Server aus verschiedenen Gründen ausfällt, kann das System darauf reagieren, indem es allen Clients dies per *Stop*-Message mitteilt. Auf die Probleme in den anderen Fällen (1., 2. und 4.) wird in dieser Arbeit wie folgt eingegangen:
Normalerweise trifft eine Rückantwort an den Server nicht innerhalb einer vorgegebenen Zeit. Trifft die Rückantwort nicht innerhalb der vorgegebenen Zeitschranke ein, so wird die Nachricht erneut gesendet und die Zeitschranke neu gesetzt. Aber davor wird zunächst ein Netzwerktest durchgeführt, indem man an Ziel-Client oder Server ein Ping sendet. Auf diese Weise wird die Netzwerkverbindung zwischen Server und Client getestet. Wenn das Testergebnis positiv ist, können anhand von Quittierungsverfahren weitere Maßnahmen wie z.B. Neuübertragung getroffen werden. Ansonsten geht man davon aus, dass es unmöglich ist, den Client oder Server zu erreichen. Eine unzuverlässige Interaktion übergibt die Nachricht nur dem Netz und es gibt keine Garantie, dass die Nachricht beim Empfänger ankommt. Die Anforderungsnachricht kommt nicht oder höchstens einmal beim Server oder Client an [TS03], [MBW08]. Um das zu vermeiden, wurde in RemotingHub ein zuverlässiger Nachrichtentransport implementiert. Dieser wird im nächsten Abschnitt weiter diskutiert.

6.6. Zuverlässiger Nachrichtentransport

Bei einem zuverlässigen Nachrichtentransport werden die Daten in der richtigen Reihenfolge, vollständig und ohne Fehler übertragen. Weiterhin müssen aber Duplikate unbedingt vermieden werden. Deswegen erfordert ein zuverlässiger Nachrichtentransport geeignete Protokollmechanismen zur Fehlererkennung und Fehlerbehebung, eine Empfänger-Bestätigung und eine Möglichkeit der Neuübertragung von Nachrichten [Ben02]. Über die mögliche Fehlererkennung wurde schon im Abschnitt 6.5 gesprochen. Im Abschnitt 6.4.3 wurde ein allgemeiner Überblick über den Nachrichtentransport in RemotingHub gegeben. Jedoch wurde auf eine wichtige Eigenschaft des Systems, nämlich die Zuverlässigkeit des Nachrichtentransports nicht näher eingegangen; dies soll daher in diesem Abschnitt erfolgen, indem in diesem Abschnitt auf die letzteren beiden o.g. Mechanismen eingegangen wird.

Bei der zuverlässigen Nachrichtenübertragung in RemotingHub wird der Server nicht blockiert, bis die Rückantwort innerhalb einer vorgegebenen Zeit eintrifft. Trifft die Rückantwort nicht innerhalb der vorgegebenen Zeitschranke ein, so wird die Nachricht erneut gesendet und die Zeitschranke neu gesetzt [TS03]. Führt das nach maximaler Anzahl der Neuübertragungen nicht zum Erfolg, so ist für den Moment kein Senden möglich (die Leitung ist entweder gestört und die Pakete gehen verloren, oder der Empfänger ist nicht empfangsbereit bzw. antwortet nicht).

Da die Gefahr besteht, dass ein Empfänger durch mehrfaches Senden die gleiche Nachricht mehrmals erhält, wird im RemotingHub systemweit die gleiche Nachrichtenidentifikation verwendet. Obwohl der Server die erneut beantwortete gleiche Nachricht mehrfach bekommt, kann er sicherstellen, dass die eingehende Anforderung mindestens einmal bearbeitet wird. Dazu hat der Server eine Nachrichtenliste, welche die bisher gesendeten Anforderungen enthält. Jedes Mal, wenn eine neue Nachricht eintrifft, stellt der Server mit Hilfe der Nachrichtenidentifikation fest, ob schon die gleiche Anforderung in der Liste steht. Trifft dies zu, so wird die Nachricht mehrfach gesendet. Ist die Nachricht noch nicht in der Liste vermerkt, so wird sie in die Liste eingetragen und anschließend zu den Abonnenten weitergeleitet. Es wird eine Bestätigung an den Source-Client gesendet, bevor eine entsprechende Rückantwort gesendet wird. Wenn es später die vom Server gesendete Rückantwort bestätigt wird, kann die Nachricht aus der Liste gestrichen werden.
Für eine zuverlässige Interaktion kann entweder [Ben02]:

1. jede Nachrichtenübertragung durch senden mit einer Rückantwort quittiert werden oder
2. ein Request und ein Reply werden zusammen durch eine Rückantwort quittiert.

Im ersten Fall muss der Server nach dem Senden der Anforderung an den Client eine Quittierung zurückschicken. Eine Rückantwort vom Server an den Client wird dann vom Client an den Server quittiert. Damit braucht ein Request mit anschließendem Reply vier Nachrichtenübertragungen. Da RemotingHub die Anforderung nicht selbst verwendet, sondern an einen weiteren Client (einen möglichen Abonnent von dieser Art Message) weiterleitet, werden noch weitere vier Nachrichtenübertragungen benötigt. Im zweiten Fall wird die Client-Server-Kommunikation als eine Einheit betrachtet, die quittiert wird. Der Client blockiert dabei, bis die Rückantwort eintrifft und diese Rückantwort wird quittiert. Nach der Anforderung soll RemotingHub eine asynchrone Kommunikation zwischen den Clients ermöglichen. Deswegen konnte der zweite Fall im diesem System nicht eingesetzt werden, sondern wurde ein hybrides Quittierungsverfahren entwickelt.
In Abbildung 6.6 ist der Ablauf des Nachrichtentransports in RemotingHub dargestellt. In Bild 1 in dieser Abbildung sendet der EAA (EA Adapter) zunächst eine Nachricht an den Server. Die Nachricht wird in die Nachrichtenliste in Tabelle 6.1 eingetragen. Um sicher zu gehen, dass die Nachricht ans Ziel gesendet wird, wird noch eine Liste für Quittierungen (ackListe) erstellt. Darin werden alle vom Server gesendeten Nachrichten gespeichert und in einem zweiten Schritt weiter an die Clients weitergeleitet. Sobald eine Bestätigungsnachricht ankommt, wird sie aus ackListe gelöscht. Wenn alle Clients die Empfangsbestätigungen für diese Nachricht gesendet haben, dann wird die Nachricht als gesendet in der Nachrichtenliste markiert und der Server sendet eine Quittierung an die EAA zurück. Somit endet die Quittierungsphase. In Bild 2 rechts werden die Clients senden die Rückantworten. Wenn alle Clients die Nachricht beantwortet haben, dann wird sie als beantwortet markiert. Nach der Weiterleitung der Rückantwort wartet der Server noch auf die Bestätigung von Client EAA, bis der Client, der die Rückantwort bekommen hat, die Empfangsbestätigung für die Rückantwort der Nachricht sendet. Sobald sie da ist, wird die Nachricht endgültig aus der Nachrichtenliste gelöscht.
Der ganze Vorgang wird aber noch komplizierter, wenn einer der Clients, wie in Abbildung 6.7 gezeigt ist, noch weitere Anforderungsnachrichten an den Server sendet. In diesem Fall wird der Ablauf wie folgt aussehen: Wenn die Anforderungsnachricht (in Schritt 5) von EAA, wie bereits im letzten Beispiel im Zusammenhang mit Abbildung 6.6 gezeigt, normal von RE

nach Server gesendet wird, dann fängt die nächste Phase an, die eigentlich genauso wie vorher abläuft. In diesem Fall wird angenommen, dass zur Verarbeitung dieser Nachricht der RE weitere Traceability-Link Informationen von DB (Datenbank) braucht. Somit sendet eine weitere Anforderung an den Server. Der Server nimmt die Nachricht auch in die Nachrichtenliste von Tabelle 6.1 auf, sucht nach den Abonnenten und leitet sie in Schritt 6 an DB weiter. In Schritt 10 bekommt der RE seine Antwort und bestätigt sie in Schritt 11. Nach solchen Nachrichtenübertragungen bekommt der EAA seine Antwort in Schritt 14 und in Schritt 15 wird die ganze Nachrichtenübermittlung abgeschlossen. Zu bemerken ist hier, dass noch ein Client Logger („Log “ in den Abbildungen) existiert. Der Client Logger ist bei jeder Art Nachrichten, die über das System laufen, automatisch abonniert, d.h er bekommt alles mit, was im System passiert.

Eventname	EventID	EvSCR	EvDest	Status	Req/Res	STime	WTime	Ansrd
CreateLink	84394	EA1	RE1	sent	Req	23:34	110	No
GetLinks	32940	RE1	DB	sent	Req	23:35	110	No
CreateLink	430940	RE1	DB	sent	Req	23:34	110	No
CreateLink	84394	RE1	EA1	sent	Res	23:34	110	Yes

Tabelle 6.1.: Eine Nachrichtenliste im Server

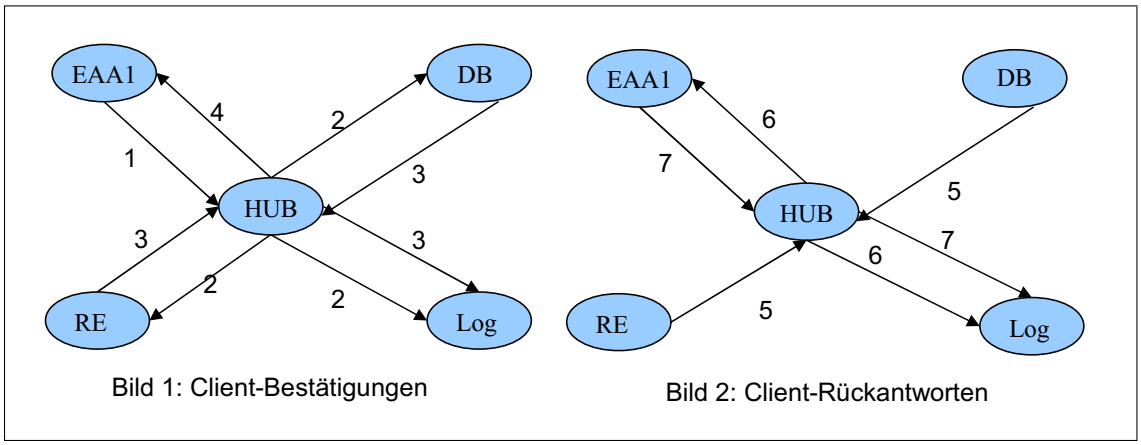

Abbildung 6.6.: Zuverlässige Nachrichtentransport im RemotingHub (einfacher Fall)

6.7. Server – Windowsdienst

Da RemotingHub einen wichtigen Dienst anbietet, nämlich die Weiterleitung von Nachrichten an weitere Abonnenten, stellt sie einen wichtigen Hauptknotenpunkt im Nachrichtenverkehr

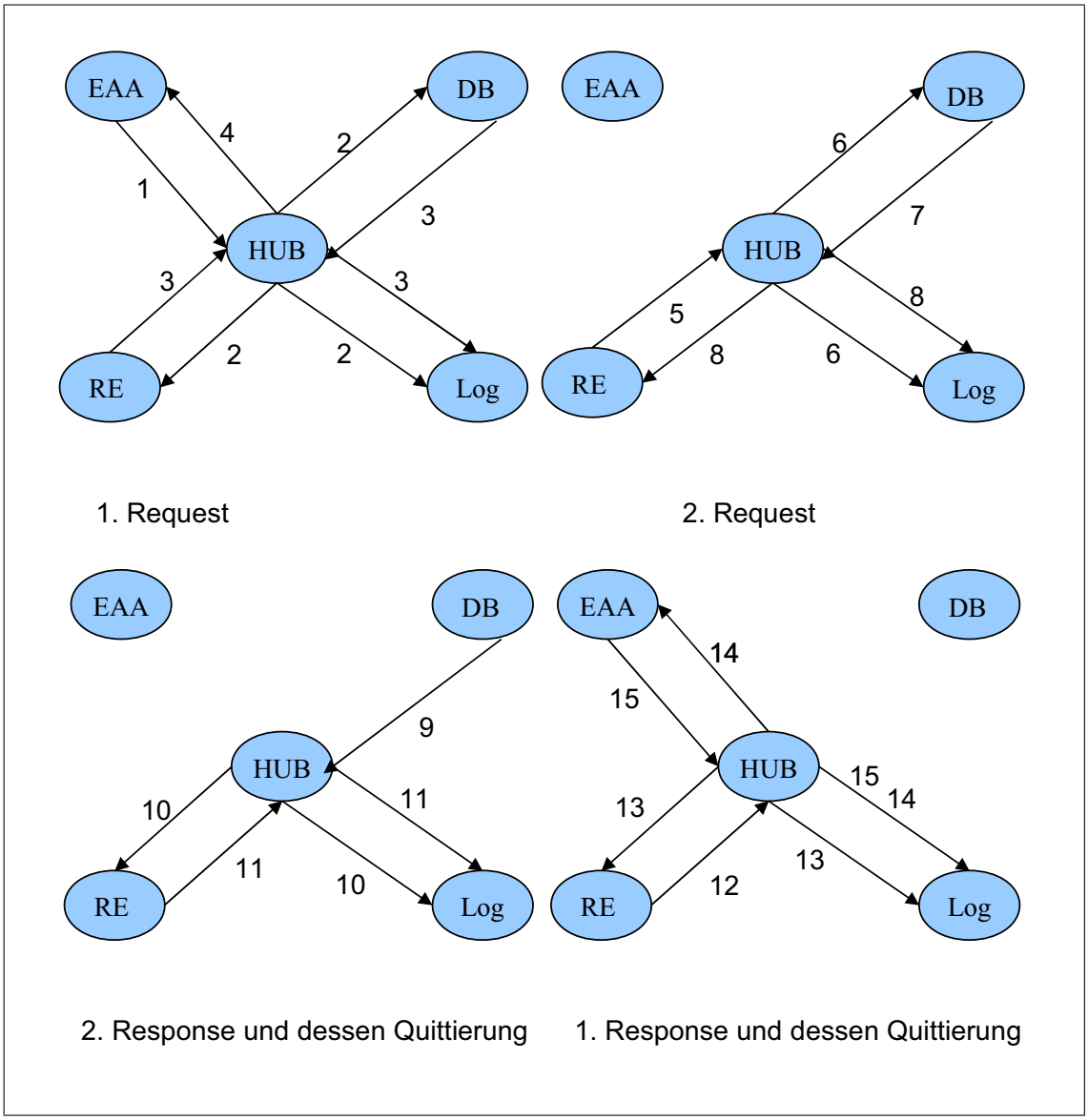

Abbildung 6.7.: Zuverlässige Nachrichtentransport im RemotingHub (komplizierter Fall)

zwischen den SW-Komponenten der verteilten TraceMaintainer-Anwendung dar. Deswegen muss sie immer erreichbar bleiben. Aus diesem Grund wurde dieser Server, RemotingHub, als Windowsdienst entwickelt, zumal eine solche Lösung zur Bereitstellung der anständigen Dienste eingesetzt wird. Mit Windows-Diensten [MSDb], [Rot03] lassen sich die ausführbaren Anwendungen mit langer Laufzeit erstellen, die in eigenen Windows-Sitzungen auszuführen sind. Solche Dienste können automatisch gestartet werden, sobald der Rechner gestartet wird. Sie können angehalten und neu gestartet werden. Es ist jedoch keine Benutzeroberfläche zu sehen. Deswegen sind Windowsdienste mit diesen Features eine ideale Wahl für die Verwendung auf einem Server. Sie sind darüber hinaus für alle Fälle geeignet, in denen Funktionen mit langer Laufzeit benötigt werden und Benutzer, die am gleichen Computer arbeiten, nicht gestört werden sollen. Deswegen ist es sinnvoller, RemotingHub in einen Windowsdienst zu integrieren.

7. Systemtest und Auswertung

Das Ziel der Softwaretechnik ist es, Qualitätssoftware zu erstellen. Der Softwaretest ist einer der wichtigen letzten Kernpunkte der SW-Entwicklung und ist ein Test des Prototyps, der in den früheren Phasen konzeptioniert und realisiert wurde. Ein entscheidender Faktor für die Qualität ist jedoch der effektive Softwaretest [PR90]. Der Begriff Softwaretest wurde von [Mye00] so definiert:

> „Testen einer Software ist der Prozess, ein Programm mit der Absicht auszuführen, den Fehler zu finden. “

Der Testprozess ist ein integraler Bestandteil jedes Qualitätsprogramms. Um sicherzustellen, dass ein Produkt in beliebiger Phase seines Lebenszyklus den Anforderungen entspricht, muss es vor dem Hintergrund dieser Anforderungen getestet werden. Aus der Definition im [Mye00] kann man sicherlich ableiten, dass der Softwaretest nicht nur die Qualität der ausgelieferten Softwareprodukte verbessert, sondern auch die Zufriedenheit der Anwender steigert und darüber hinaus die Wartungskosten senkt. Im Gegensatz dazu führt der fehlende und ineffektive Test zu negativen Ergebnissen wie schlechtere Qualität der Produkte, Unzufriedenheit der Kunden, unzuverlässige und ungenaue Ergebnisse. Ganz im Sinne von traditionellem Softwaretest steht in dieser Arbeit das Testen des hier realisierten Systems, RemotingHub, im Mittelpunkt.
Obwohl RemotingHub bis jetzt einigermaßen fehlerfrei die eingehenden Nachrichten automatisch verarbeiten und Systemanforderungen erfüllen kann, bedeutet dies noch lange nicht, dass es dauerhaft fehlerfrei arbeiten kann. Man muss deshalb die Zuverlässigkeit und Anwendbarkeit dieser Prototyp-Software vor dem richtigen Einsatz verschiedenen Testverfahren unterziehen. Im Rahmen dieser Arbeit musste als Grundlage für solche Tests eine geeignete Testumgebung gewählt werden, in die der Prototyp integriert werden konnte. Diese soll im Folgenden vorgestellt werden. Ziel ist es, die Testumgebung vollständig lauffähig umzusetzen, um später erste experimentelle Evaluierungsaussagen zur Leistungsfähigkeit und Zuverlässigkeit des Prototyps treffen zu können.

7.1. Systemtest mit analogen Clients

Wie bereits in der Zielsetzung beschrieben, wird das System in eine Testumgebung integriert und parallel entwickelt. In diesem Zusammenhang spricht von einem White-Box-Test. Aber das Testziel war, RemotingHub als eine Black-Box zu betrachten und auf die Systemfunktionalitäten zu testen. So möchte man feststellen, ob das System die Systemanforderungen korrekt umgesetzt hat. Dazu gehört, die korrekten Nachrichten an die korrekten Clients weiterleiten zu können bzw. auf die Ausnahmen korrekt reagieren zu können.
In diesem Sinne kann man sich vorstellen, dass zum Test von RemotingHub sowohl White-Box-

Test als auch Black-Box-Test [PR90] verwendet wurde . Zu Beginn dieser Arbeit waren nicht alle Komponenten des TraceMaintainer zur Entwicklung einer verteilten Anwendung geeignet (zumindest für hier zu konzipierende und realisierende Nachrichtentransportsystem), und die Anpassungen im TraceMaintainer konnten nicht rechtzeitig bereitgestellt werden. Daher mussten im Laufe der Entwicklung des Systems bestimmte analoge Clients konzipiert werden und eine problemspezifische Testumgebung erstellt werden, um den Prototyp des Systems ohne Problem zu testen. Tatsächlich wurden solche analogen Clients realisiert, sodass man sich dabei wirklich eine reale verteilte TraceMaintainer-Anwendung vorstellen kann. Die Programmierung erfolgte in der Sprache C# mithilfe von Visual Studio 2008 und es wurde auch .Net Framework Version 3.5 verwendet.

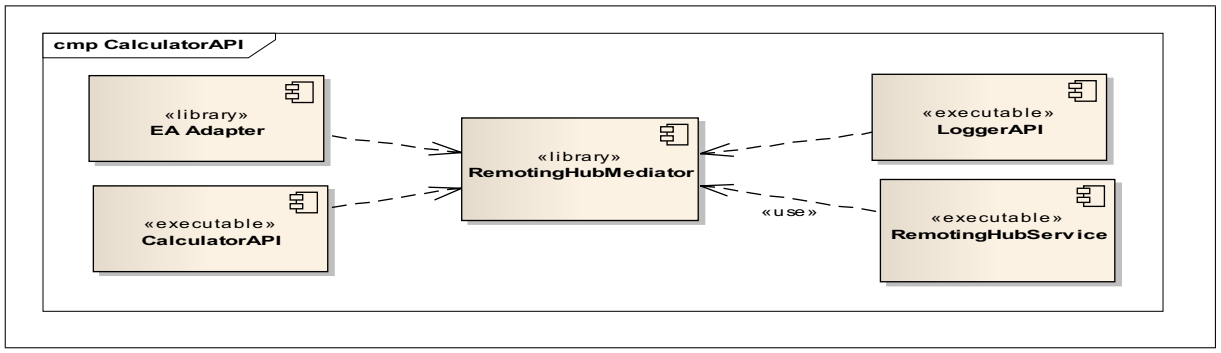

Abbildung 7.1.: Die Testumgebung für RemotingHub

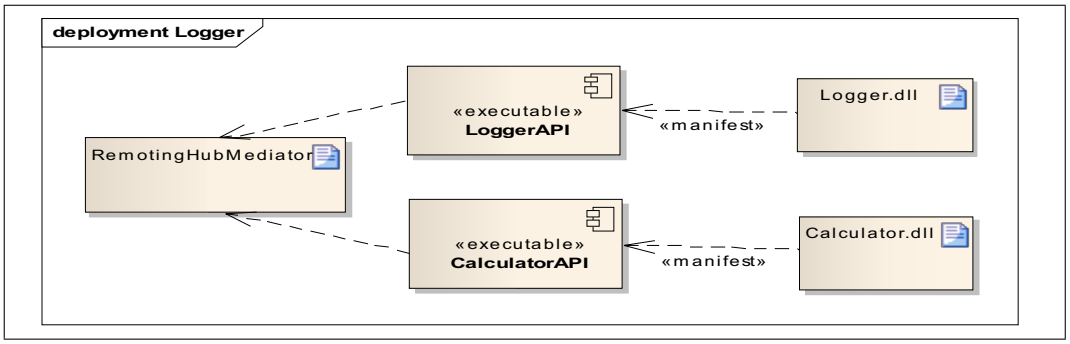

Abbildung 7.2.: die physische Struktur der Clients

Da das eigentliche Hindernis die starke Abhängigkeit von TS und RE in der TM-Anwendung war, konnte man sie bei der Entwicklung von RemotingHub nicht zum Testen einsetzen und demzufolge müssen diese Komponenten unbedingt durch eine analoge SW-Komponente in der Testumgebung ersetzt und getestet werden. Wie Abbildung 7.1 zeigt, kam als Ersatz für diese beiden Komponenten eine einfache Calculator-Anwendung zum Einsatz. Die Clients sind ein

EA-Adapter (EAA), eine Calculator-Anwendung und ein Logger. Der EA-Adapter ist ebenso wie Eventgenerator in EA integriert und spielt hier die Rolle von Eventgenerator. Der Logger dient dazu, die Ereignisse auf dem Server zu protokollieren. Wie in dem Verteilungsdiagramm von Abbildung 7.2 gezeigt ist, bestehen die Clients, Logger und CalculatorAPI physikalisch aus zwei Teilen - einem ausführbaren Teil und einer Klassenbibliothek. Da es geplant war, die RE und TS zukünftig auch in dieser Form in RemotingHub einzuführen, wurden die beiden analogen Clients in derselben Art und Weise gebaut. Noch eine Besonderheit ist zu betonen, und zwar dass alle dieser Clients die abstrakte Klasse *ClientConnection* im *RemotingHubMediator* implementieren müssen, um überhaupt ein Proxyobjekt erzeugen und danach eine Verbindung zum Server herstellen zu können. Auf Abbildung 7.3 ist ein Deployment-Diagramm (Verteilungsdiagramm) zu sehen, dass wie die Clients (aktive wie passive) und Server auf den Rechnerknoten verteilt sind. Das ist jedoch keine Festlegung, sondern nur ein typisches Beispiel. Im Folgenden werden alle dieser drei Clients vorgestellt.

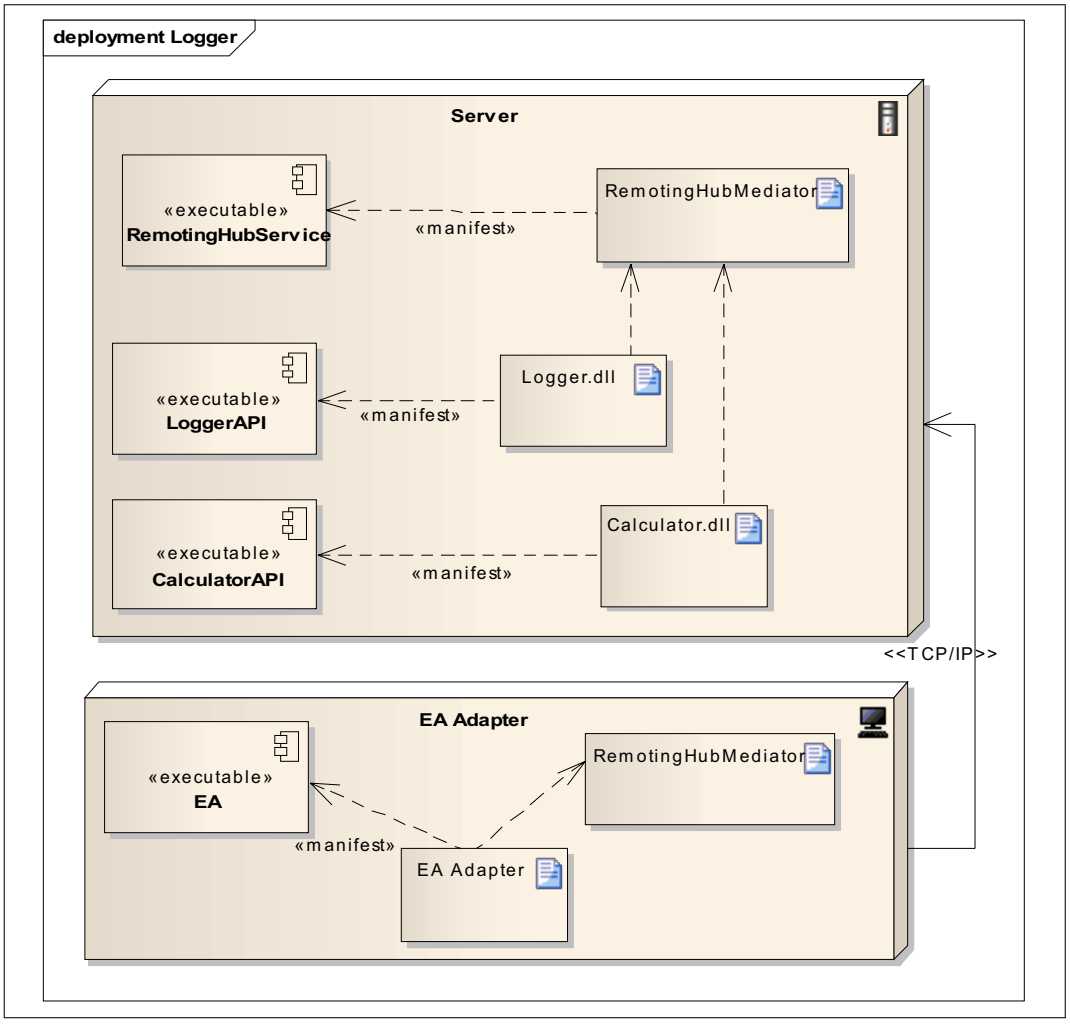

Abbildung 7.3.: Deployment-Diagramm für die Komponente der Testumgebung

7.2. Logger ist ein eigenständiger Client

Ein Logger in dieser Testumgebung ist ein Client, der ganze Ereignisse auf dem Server erfassen muss. Die Ereignisse umfassen die An- und Abmeldung eines Clients, die Client Registerinformation, verschiedene Fehler und Ausnahmen, die Aufnahmen in die Nachrichtenliste, deren Weiterleitung und Löschung, ... usw. Der Logger ist während der Systementwicklung nur testweise im System eingesetzt. Es könnte in Zukunft möglich sein, dass man einen Logger als festen Bestandteil von RemotingHub auf Dauer einsetzt.
Die Hauptidee ist, mit der angegebenen Pfadangabe ein Log-File zu erzeugen. Der Filename bildet sich aus dem Wort „LogFile “ und dem aktuellen Datum, nach dem Muster LogFile_Tag_Monat_Jahr.txt. Wenn das File in dem angegebenen Ordner nicht existiert, dann wird ein neues File erzeugt, ansonsten wird das vorhandene LogFile aktualisiert. Die Idee mit dem Filenamen ist, täglich einmal ein neues Logfile zu erzeugen, um die Systemereignisse getrennt zu protokollieren. In dem Logfile werden alle Einträge mit genauen Orts- und Zeitangaben protokolliert.
RemotingHub kann in der Regel mehrere Sessions verwalten und wurde besonders zu diesem Zweck entwickelt. Es wurde in den letzten Kapiteln erläutert, dass eine minimale Session außer dem Server noch aus RE, TS und EA-Adapter besteht. In der Testumgebung kommt noch der Logger als eigenständige SW-Komponente hinzu. Die Besonderheit beim Logger ist, dass mindestens ein Logger in jeder Session existieren muss. Aber es ist nicht zweckmäßig, für jede Session ein getrenntes LogFile zu erzeugen, sondern vielmehr müssen alle Ereignisse, die im Server vorkommen, in ein LogFile eingeschrieben werden. Dazu muss man sich einen besonderen Trick überlegen. Und dieser Trick lautet ein in der Softwareentwicklung eingesetztes Entwurfsmuster - *Singleton* [Ker08].
Singleton gehört zur Kategorie der Erzeugungsmuster. Es verhindert, dass von einer Klasse mehr als ein Objekt erzeugt werden kann. Beim Singleton-Pattern, einem Entwurfsmuster, geht es um die Anzahl der Instanzen einer Klasse. Mit Hilfe des *Singleton*-Pattern wird sichergestellt, dass von einer Klasse nur jeweils eine Instanz erstellt werden kann. Dieses Pattern verhindert das Erstellen von Instanzen durch den Konstruktor. Der Trick besteht nun aber darin, den Konstruktor „private “ zu setzen, sodass dieser nur intern verwendet werden kann. Zudem braucht es einen statischen Member in der Klasse, der den Status der Instanz hält. Um eine Instanz zu bekommen, wird eine öffentliche Methode implementiert. In dieser wird ein Objekt der Klasse erstellt, und zurückgegeben, wenn das Objekt noch nicht vorhanden ist. Im Folgenden ist ein Beispiel zu sehen.

```
public sealed class MySingletonClass
{
   static MySingletonClass instance=null;
  static readonly object padlock = new object();
   MySingletonClass()
  {
  }
  public static MySingletonClass Instance
  {
    get
```

```
        {
          lock (padlock)
          {
             if (instance==null)
             {
              instance = new MySingletonClass();
              }
            return instance;
          }
        }
      }
    }
```

Auf diese Weise sichert das *Singleton*-Pattern ab, dass von der Logger-Klasse nur eine Instanz existieren kann. Diese wird über die öffentliche Methode „*getInstance*“ der Klasse MySingletonClass erzeugt. Somit können die verschiedenen Logger-Clients von einer bestimmten Session in ein einziges LogFile alles hineinschreiben.

7.3. Calculator Anwendung

Wie in Abschnitt 7.1 beschrieben ist, steht die Calculator-Anwendung symbolisch für die Anwendungen von RE und TS. Da der Kern von TraceMaintainer, der RE, ohne dessen Datenbank TraceStore nicht arbeitsfähig ist, wurden hier symbolisch die beiden Komponenten zu einer Komponenten, nämlich Calculator-Anwendung, zusammengefasst. In der eigentlichen TraceMaintainer-Anwendung bekommt der RE die Ereignisse vom Eventgenerator übergeben. In dieser Testumgebung sendet der EA-Adapter auch die Ereignisse bzw. Nachrichten über den Server an die Calculator-Anwendung. Die Übermittlung dieser Nachrichten erfolgt nach den Nachrichtentransportmethoden, die im letzten Kapitel vorgestellt wurden.
Die in der Testumgebung eingesetzte Calculator-Anwendung ist in der Lage, einfache arithmetische Aufgaben wie in Abbildung 7.4 zu erledigen. Jede Nachricht, die bei Calculator-Anwendung angekommen ist, muss ein Format wie Event haben, was in Anhang A.3 ersichtlich ist. Wenn die hier empfangene Nachricht eine Anforderung ist, dann muss der Calculator unbedingt darauf reagieren. Im Klartext bedeutet, dass wenn es sich hier um eine Anforderungsnachricht (der Nachrichtentyp ist Request) handelt, muss an den Server unbedingt eine Quittierung geschickt werden. Dazu wird zunächst die auszuführende Methode anhand des Nachrichtennamen mit Hilfe von *System.Reflection* gewonnen. *System.Reflection* ist ein Namensraum in der .Net-Framework-Klassenbibliothek, die in .NET seit der Version 1.0 vorhanden ist. *Reflection* bietet Objekte vom Typ Type an, die in einer Assembly eingekapselt ist. Man kann mit Hilfe von Reflection den Typ von einem vorhandenem Objekt bestimmen und somit dessen Methoden aufrufen oder auf dessen Eigenschaften zugreifen. Dank Reflection kann man über alle Eigenschaften der aktuellen Klasse, *Rechner* iterieren und etwas ausrechnen lassen. Wenn das Ergebnis da ist, kann man es wieder in Form einer Nachricht als Antwort (Response) an den Server zurückschicken.

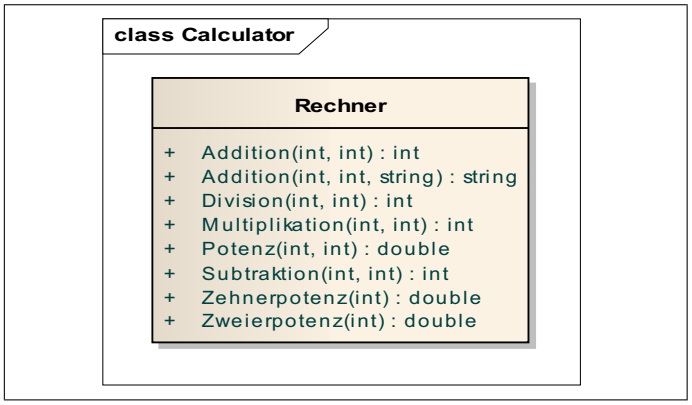

Abbildung 7.4.: die Rechnerfunktionen der Calculator-Anwendung

7.4. Der EA-Adapter

Im Gegensatz zu den vorherigen beiden Clients ist der EA-Adapter ein aktiver Client. Dank dieser Clients werden im Prototype die Sessions auf dem Server gebildet, wenn seine gelieferten Informationen die im Abschnitt 6.3 beschriebene Anmeldungsregel bestehen. Außerdem ist er zur Zeit in der Testumgebung der einzige Client, der aktiv Anforderungsnachrichten an den Server schicken kann. Ein weiterer Unterschied zu den vorherigen beiden Clients ist, dass der EA-Adapter keine eigenständig ausführbarer Client, sondern eine COM-Interoperable Klassenbibliothek ist, die von Enterprise Architect (EA) gestartet und verwendet wird. Um diesen Client anwenden zu können, müssen also folgende Anforderungen unbedingt erfüllt werden [Sys98].

1. EA muss auf dem aktuellen Rechner installiert werden.
2. Man muss noch einen neuen Eintrag in Windows Registry machen, damit EA die Anwesenheit dieses *AddIns* erkennen kann und den Einstiegspunkt in diese Klassenbibliothek finden kann.
3. Da der EA-Adapter eine COM-Interoperable Klassenbibliothek ist, muss er zunächst unbedingt mit dem entsprechenden Tool (*Regsvr32* bzw. *RegAsm*) registriert werden.

Die Realisierung des EA-Adapters erfolgte auf Basis von einem auf der EA-Website angebotenen Template zur Erzeugung eines internen Add-INs. Dieses Template ist eine Automatisierungsschnittstelle, die nur für die Programmiersprache C# angeboten wird. Aus dem Klassendiagramm in Anhang 5 kann man ersehen, dass es sich um eine relativ einfache Windowsanwendung handelt. Die Klasse *Main* ist in dem EA-Template bereits enthalten. Für die EA ist sie der einzige Einstiegspunkt zur Laufzeit in den EA-Adapter. Die Klasse *EAAdapterConnection*,

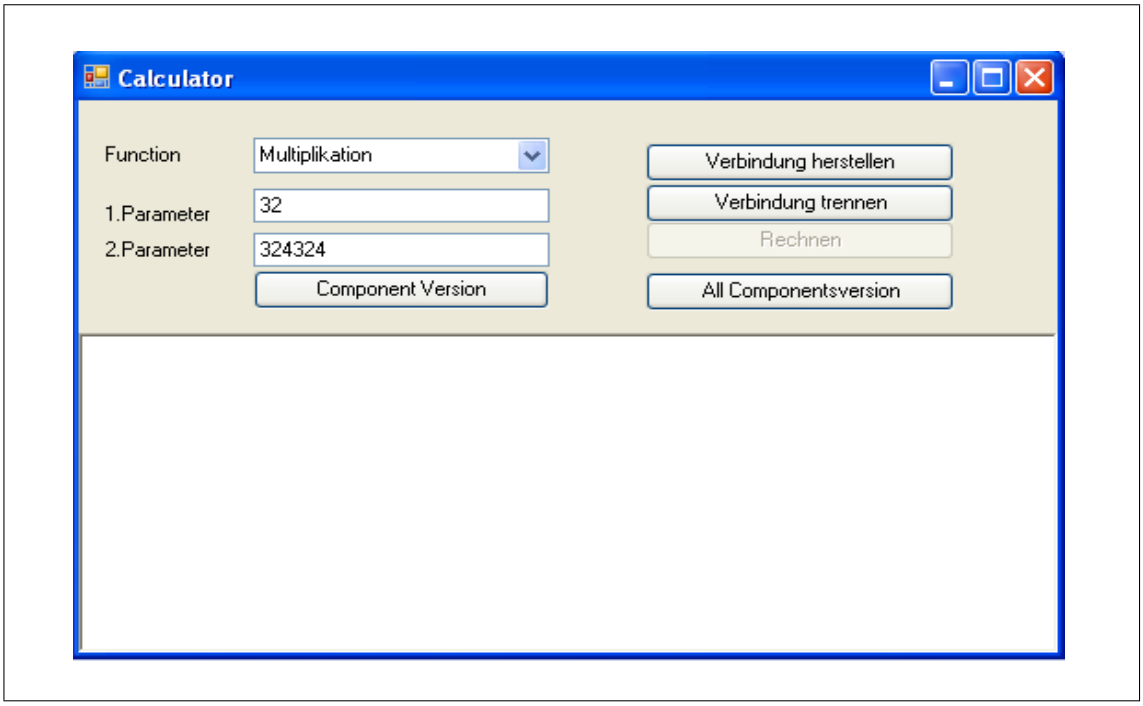

Abbildung 7.5.: Bedienoberfläche für die Testumgebung des RemotingHubs

wie der Name schon sagt, stellt die Verbindung zum Server her. Bei der Registrierung muss sie die nötigen Informationen wie *Hostname*, aktueller *Username* und*Modellname*, *Tool-Typ*, ... usw. liefern. Wenn die Anmeldung erfolgreich abgeschlossen ist, kann sie Nachrichten an den Server schicken und Nachrichten aus dem Server bekommen. Die empfangene Nachrichten werden teilweise verarbeitet und wird nach der Klasse *Form1* gesendet. Damit man die Testumgebung einfach bedienen kann, wurde die letzte Klasse *Form1* realisiert. Wie man in Abbildung 7.5 sieht, hat sie eine relativ verständliche Bedienoberfläche. Wie die Beschriftungen auf den Buttons schon sagen, lässt sich hier eine Verbindung zum Server herstellen und wieder trennen. Außerdem bietet der EA-Adapter noch Abfragemöglichkeiten nach den Versionen aller beteiligten Clients in der aktuellen Session. Nach Herstellung der Verbindung zum Server lässt sich die Rechnerfunktionen testen. Dafür muss man zunächst eine Rechnerfunktion aus der Kombo-Box auswählen, entsprechend Parameter eingeben und rechnen lassen. Wenn das Ergebnis bzw. andere Nachrichten wie Warnungen oder Ausnahmen vom Server zurückgesendet werden, ist das sofort im unteren Teil des Fensters zu sehen. Wenn man hier die aktuelle Verbindung trennt, wird zunächst eine Stop-Message (z.B. „Achtung: EA-Adapter möchte jetzt aus der aktuellen Session zurücktreten “) an den Server gesendet, damit der Server entsprechende Maßnahmen einleiten kann. Zum Schluss werden das aktuelle Proxyobjekt und das Callback-Objekt zerstört, somit endet der Test.
Es muss hier erwähnt werden, dass die Realisierung des EA-Adapter in die Testumgebung im Vergleich zu den Calculator-Anwendung und Logger sehr kompliziert war. Es war nicht so einfach gewesen eine bidirektionale Kommmunikation zu erstellen. Es konnten mit vorhandem Template für Automatisierungsschnittstelle von EA aus an den Server ohne grosse Schwerigkeiten Nachrichten geschickt werden. Das Problem ist jedoch, dass EA bzw. EA-Adapter die Rückantworten vom Server ingnoriert bzw. nicht empfangen kann. Da Enterprise Architect ei-

ne komplett eigenständige SW ist, war es sehr schwerig gewesen, ohne jegliche Unterstützung von Firma *SPARX Systeme* die Problemursachen heraus zu finden. Deswegen wollte man das Problem mit dem Ansatz durch Polling Verfahren lösen. Wie im Abschnitt 5.7.1 vorgestellt, ist das Polling verfahren ist negativ zu bewerten. Außerdem wurde dieser Lösungsansatz vom Betreuer gründsätzlich abgelehnt. Nach zwei wöchigen Recherche im Internet und über die Programmcodes, ist endlich gelungen, die Rückantworten vom Server an EA-Adapter zu senden. Die Ursache von dem Problem war, dass die assembly *RemotingHubMediator* in den Hauptspeicher ausdrücklich nochmals nachgeladen werden muss.

```
System.Reflection.Assembly CurrentDomain_AssemblyResolve(object sender, Re-
solveEventArgs args)
{
Assembly result = Assembly.LoadFrom(@ „<kompltter pfad von RemotingHub-
Mediator.dll>“);
return result;
}
```

Der Ursache ist unklar. Laut *Memorydump* ist genau dieselbe *DLL* bereits im speicher vorhanden, bevor die oben genannte methode aufgerufen wird. Laut Modulesfenster im Visual Studio 2008 ist diese auch in der richtigen version geladen. Anhand Ausgabefenster ist diese sogar bereits vor der addinframework *DLL* geladen. Da die Ursache dieses Problems mit dem eigentlichen Ziel der Arbeit irrelevant ist, wurde nicht weiterhin tiefgehend untersucht.

7.5. Evaluation der Testumgebung

Der Softwaretest ist in jeder Phase der Softwareentwicklung ein sehr wichtiges Thema. Effektiv ausgeführte Softwaretests werden nicht nur die Qualität der Software steigern, sondern auch die Benutzer zufrieden stellen und die Entwicklungskosten der Software senken. Darüber hinaus können sie auch zur Präzision und Verlässlichkeit von Produkten beitragen [Mye00]. In allen maßgebenden Normen zur Erstellung von Software wird ein System zur Erfassung und Verfolgung der Fehler benötigt. Das ist auch notwendig, denn bereits bei mittleren Projekten kommt man im Laufe der Jahre auf ein paar Tausend Fehler und Änderungen. Nur sollte man bei diesen Zahlen nicht erschrecken, weil der gefundene Fehler stets ein guter Fehler ist [PR90]. In diesem Sinne hat man auch geplant, eine Testumgebung für RemotingHub zu entwickeln. Wie in den letzten Abschnitten beschrieben ist, wurde diese Testumgebung im Laufe der Systementwicklung parallel realisiert und hat dazu gedient, aus der Sicht des Entwicklers alle Funktionalitäten des Systems zu testen. Das Hauptziel war, die Anwendbarkeit und Zuverlässigkeit des Prototyps experimentell zu testen.
Nachdem man diese Testumgebung zum Experimentieren eingerichtet hat, wurde von vielen Interessierten zahlreiche Tests durchgeführt. Darüber hinaus wurde das Testprogramm mehrmals vor dem Betreuer und anderen Mitarbeitern demonstriert und die Testidee erklärt. Sie haben die Arbeiten des Testprogramms überwacht und konstruktive Meinungen geäußert. Alle erstellten Nachrichten in der Testumgebung wurden mehrmals gezielt in verschiedenen Ausgangssituationen getestet. Während der Experimentierung hatte das Testprogramm viele Fehler gefunden, die während der Systementwicklung leicht zu übersehen waren. Diese Fehler wurden zeitgleich

im System zurückverfolgt und korrigiert.
Im Rahmen der Evaluierung des Systems entsteht eine Hauptseminarbeit, welche durch Christian Kittler ausgearbeitet wird. Er wird eine Evaluierung des RemotingHub bezüglich Verarbeitungzeit und Nachrichtendauer durchführen. Er integriert RemotingHub in die eigentliche TM-Anwendung, wobei besonderer Wert auf die Nutzung von verteiler Kommunikation der SW-Komponente der TM-Anwendung gelegt wird. Die RuleEngine wird dabei so konfiguriert, dass sie einkommende "Modeleventsëinfach als Response mit dem EventName "LinkUpdateßurücksendet. Der Eventgenerator erzeugt eigene LogFiles für versendete und empfange Events zu protokollieren. Auf diese Weise kann man genau verfolgen, wielange der Versand durch den RemotingHub dauerte. Mit Hilfe vorhandener LogFiles kann ebenfalls ein Stresstest des RemotingHub durchgeführt werden. Wenn diese Arbeit abgeschlossen ist, können noch genauere Aussagen zur Evaluierung von RemotingHub gemacht werden.

Obwohl das System heute das Systemanforderungen erfüllt, bedeutet dies nicht, dass es fehlerfrei und in jeder Situation zuverlässig angewendet werden kann. Es sind also immer noch umfangreiche Experimente über einen längeren Zeitraum nötig, um sichere und genaue Aussagen über die Zuverlässigkeit und Anwendbarkeit dieses Prototyps zu machen, was im Rahmen dieser Diplomarbeit leider nicht möglich ist. Trotz alledem soll der Prozess im Anschluss an diese Arbeit weitergeführt werden.

8. Zusammenfassung und Ausblick

Der TraceMaintainer ist eine softwaretechnische Realisierung der Relationsänderungen zwischen Analyse- und Designobjekten. Die dabei erreichbare automatische bzw. teilautomatische Nachführung von Traceability-Link-Beziehungen [MGP08], die im Normalfall eigentlich einen sehr hohen manuellen Aufwand bei der SW-Validierung und -Verifizierung erfordert, ermöglicht eine Reduzierung des Arbeitsaufwands. Doch darüber hinaus schließt er im Umgang mit der Nachführung der Traceability-Links vielmehr eine große Lücke, nämlich die mangelnde technische Unterstützung aktueller SW-Entwicklungswerkzeuge.
Der TraceMaintainer wird zur Zeit lediglich im lokalen Rechnerbereich verwendet. Die bisher existierenden Softwarekomponenten sind relativ stark voneinander abhängig und außerdem können sie nur noch lokal auf einem einzigen Rechner zusammenarbeiten. Weiterhin ist eine Erweiterbarkeit der Softwarelösungen nur schwer möglich. Um aber den gesamten Funktionsumfang des TraceMaintainer ausschöpfen zu können, muss er unbedingt in einer verteilten Rechnerumgebung zur Anwendung gebracht werden. Dafür müssen geeignete Programme zu seiner Erweiterbarkeit entwickelt werden.
Durch die Entwicklung eines geeigneten Ansatzes sollte daher gezeigt werden, dass sich dieser Aufwand reduzieren lässt und die Kommunikation der SW-Komponenten von TraceMaintainer vereinfacht werden kann. Dabei sollten Modularität und eine damit verbundene spätere Erweiterbarkeit im Vordergrund stehen. Die vorliegende Diplomarbeit zeigt die Ergebnisse der Entwicklung von Programmkomponenten im Zusammenhang mit dem Entwurf eines modularen Nachrichtentransportsystems für TraceMaintainer.

8.1. Schlussfolgerungen

Der TraceMaintainer (TM) ist keine eigenständige SW-Anwendung [MGP08]. Zur Zeit existieren drei Komponenten für unterschiedlichste Aufgaben. Damit der TM angewendet werden kann, ist außerdem noch mindestens ein UML-Modellierungstool nötig. Das hier entwickelte Verfahren erlaubt, diese Komponenten komplett eigenständig arbeiten zu lassen. So kann z.B. die RuleEngine in eine eigene Applikation ausgelagert werden. Damit man das volle Spektrum an Funktionalitäten des TraceMaintainer ausnutzen kann, ist dessen verteilte Anwendbarkeit verschiedener Funktionen im Internet notwendig. Daher wurde untersucht, welche Technologien für die Realisierung einer verteilten Anwendung in Frage kommen könnten. Als drei der derzeit wichtigen Technologien haben sich *.NET Remoting*, *WCF* und *XML-Webdienste* herausgestellt. Für die prototypische Realisierung des Systems wurde entschieden, davon lediglich .Net Remoting eingzusetzen.
Im anfänglichen TraceMaintainer wurde die RuleEngine als feste Komponente in den Eventgenerator und TraceStore integriert und über Methodenzugriffe angesteuert. Dadurch, dass die

künftigen SW-Komponenten von TraceMaintainer im Internet komplett voneinander unabhängig sind, tritt ein neues Problem auf. Ein Komponente kann die anderen gar nicht instanziieren und darauf zugreifen. Das war der Kernpunkt dieser Diplomarbeit, warum das Nachrichtentransportsystem, RemotingHub realisiert wird. Der Lösungsansatz zu diesem Problem war, ein softwareartiges, intelligentes Hub zu entwicklen, der die ursprünglichen Kommunikation zwischen Komponenten des TraceMaintainer intelligent koordiniert. Dazu wurde das Konzept *Publisher-Subscriber-Modell* eingesetzt. Nach diesem Konzept können die neuen Komponenten wie folgt klassifiziert werden, sodass jede SW-Komponente nur den RemotingHub als Kommunikationspartner kennt.

- RemotingHub ist ein Broker, der die Nachrichten verteilt.
- Alle ursprünglichen SW-Komponenten von TraceMaintainer sind entweder ein Publisher oder ein Subscriber.

Ein weiteres Problem war, mindestens eine RuleEngine und einen TraceStore für jeden Eventgeneratoren immer zur Verfügung zu stellen. In dieser Arbeit wurde dazu eine flexible, XML-basierte Regelbasis entwickelt, mit der es möglich ist, von RemotingHub bestimmte SW-Komponenten starten und stoppen zu können. Voraussetzung dafür ist, dass diese zu startenden und zu stoppenden Komponenten auf dem gleichen Rechner wie Server laufen. Der Vorteil des Verfahrens ist, dass man jederzeit diese Kriterien erweitern oder mithilfe verschiedener XML-Techniken an zukünftige Erfordernisse anpassen kann. Das Datenformat XML bietet zudem aufgrund seiner großen Verbreitung eine große Zukunftssicherheit. Darauf aufbauend wurde die wichtige Funktionalität von RemotingHub, *Sessionverwaltung*, weiterentwickelt, um die Zusammenarbeit dieser Komponenten auch nach dem Einsatz von RemotingHub zu gewährleisten. Damit die Sessionverwaltung wie gewünscht funktioniert, wurde das Konzept *TIM* (**T**raceability **I**nformation **M**odel) eingeführt. Anhand des TIM kann entschieden werden, ob geöffnet werden soll. Die Sessionverwaltung ermöglicht den Anwendern von Eventgeneratoren (EG) im Internet, sogar, verschiedene Modelle, die die gleichen Traceability-Link-Beziehungen haben, zu verarbeiten.
Um das realisierte System nach Funktionalitäten zu testen, wurde eine Testumgebung gebaut. Die ersten Ergebnisse zeigen, dass die angestrebte Umsetzung vollständig erfüllt wurde. Weiterhin bedarf es keiner großen Einführung in die Bedienung. So waren die Testpersonen schon nach kurzer Zeit in der Lage, das System zu testen. Eine Sendung von Nachrichten war dabei ebenso möglich wie das Empfangen bzw. Bearbeiten und Weiterleiten von bereits eingegangenen Nachrichten. Somit wurde im Rahmen dieser Diplomarbeit besonders gezeigt, dass die Automatisierung des Traceability-Prozesses durch den TraceMaintainer auch in der verteilten Rechnerumgebung möglich ist.
Die Realisierung des Nachrichtentransportsystems, RemotingHub war das Hauptziel dieser Diplomarbeit, das in Zukunft eine modulare Anwendung von TraceMaintainer ermöglichen soll. Der wichtige Vorteil dieser Realisierung ist, dass man den TraceMaintainer nicht nur in der verteilten Rechnerumgebung anwenden kann, sondern ihn auch einfach erweitern und mit anderen Tools kombinieren kann. Kurz gesagt, der Aufbau des TraceMaintainer wird jetzt noch mehr modular.

8.2. Weiterführende Arbeiten

Im Rahmen dieser Diplomarbeit wurde mit *RemotingHub* ein eventbasierter Nachrichtenaustausch geschaffen. Diese zentrale SW-Komponente von TM ermöglicht es, nicht nur SW-Komponenten wie RuleEngine in eine eigene Applikation auszulagern, sondern auch die ursprüngliche Kommunikation zwischen diesen SW-Komponenten sowohl im lokalen Rechnerbetrieb als auch in verteilter Rechnerumgebung zu unterstützen. Darauf aufbauend wurden in das System noch eine Reihe Funktionalitäten wie *Start-Stop Mechanismus*, *Sessionverwaltung* und *Nachrichtenverteilung* eingebaut, um die Zusammenarbeit dieser SW-Komponenten nach wie vor ohne Schwierigkeiten zu gewährleisten. Es wurde außerdem eine Testumgebung entwickelt, um RemotingHub nach den Funktionalitäten zu testen. Die Testergebnisse zeigen, dass RemotingHub alle aktuellen Anforderungen vollständig erfüllt hat. In der neuen TM-Anwendung werden z.B. die verschiedenen Events, die ausgehend von Modelländerungen durch den Eventgenerator erzeugt werden, nicht direkt an RuleEngine, sondern über RemotingHub versendet. Die Rückantworten von RuleEngine werden genauso über RemotingHub ans Ziel zurückgesendet. SW-Komponenten wie RuleEngine und TraceStore können als passive Komponenten durch RemotingHub gestartet und gestoppt werden sowie in eine Session aufgenommen und/oder wieder daraus entfernt werden.
Obwohl die bisherigen Tester mit der Funktionsweise von RemotingHub zufrieden waren, kann das System nicht komplett fehlerfrei sein. Um aber weitere Fehler zu finden, muss man aktiv und zielstrebig nach ihnen forschen und stets davon ausgehen, dass sie vorhanden sind [Tha00]. Dass eine Software in verschiedenen Ausgangspunkten getestet, evaluiert und verbessert wird, ist auch Voraussetzung für ihren realen Einsatz. Obwohl im Rahmen dieser Diplomarbeit alle relevanten Probleme bezüglich der Realisierung des Systems prinzipiell gelöst sind, sind aus Sicht des Entwicklers an folgenden Stellen jedoch immer noch Verbesserungen vorzunehmen:
Die Zentralisierung führt zur funktionalen Asymmetrie im System, die aus Fehlertoleranzgründen nicht gewünscht ist [Ben02]. Da in RemotingHub alle Kommunikationsentscheidungen, wie die Kontrolle über Nachrichtenverteilung und Sessionverwaltung, nur zentral von dem Server erledigt werden sollen, besteht die Gefahr, dass die Clients unmöglich miteinander kommunizieren können, wenn der Server ausfällt. Dies ist noch ein weiterer Punkt, an dem man über Verbesserungen nachdenken muss.
Man kann noch überlegen, RemotingHubs im Internet beliebig zu kaskadieren, um eine größere Netzausdehnung zu erreichen, was das System gegen Serverausfälle noch robuster machen könnte. Außerdem kann das Nachrichtenübermittlungsverfahren weiter optimiert werden. Der gegenwärtige Ansatz zur Nachrichtenübermittlung beruht auf den aktuellen Anforderungen an das System. Obwohl es derzeit die Systemanforderungen gut erfüllt, kann es trotzdem ineffizient oder fehleranfällig sein.
Neben der Optimierung des Nachrichtentransportverfahrens könnte die Verbesserung des gesamten Systems eine interessante Arbeit zur Weiterentwicklung sein. Denkbar ist daher auch, in das System andere Ideen wie z.B. Client-Autorisierung einzubauen, damit beispielsweise der Server-Rechner vor Computer-Schädlingen geschützt werden kann.

Literaturverzeichnis

[Ben02] Günther Bengel. *Verteilte Systeme*. Freidl, Vieweg&Sohh Verlagsgesellschaft mbH, 2002. ISBN:3-528-25738-5.

[BH94] P. Borowka and M. Hein. *Hub Systeme; Funktion.Konzeption.Einsatzgebiete*. DATACOM Verlag, 1994. ISBN:3-89238-099-6.

[BN84] A. D. Birrell and B. J. Nelson. Implementing remote procedure calls. *ACM Transactions on ComputerSystems*, 2, 1984.

[Com] Galileo Computing. *Visual C# 2008*. Galileo Computing. http://www.multidata.at/SitesMD/Doc_VisualCSharp/index.htm.

[EF03] A. Eberhart and S. Fischer. *Web Services; Grundlagen und praktische Umsetzung mit J2EE und .Net*. Carl HanserVerlag München, 2003. ISBN:3-446-22530-7.

[Exp07] FedEx Express. Hub system - das hub and spokes konzept. *Europa, Middle East and Africa*, 2007.

[GF94] O. C. Z. Gotel and A. C. W. Finkelstein. An analysis of the requirements traceability problem. 1994.

[Gmb87] LS GmbH. Hub and spoke system, 1987. http://www.lsgmbh.de/deutsch/begriff_H.htm ; letzter Zugriff: 05.02.10.

[Haa01] O. Haase. *Kommunikation in verteilten Anwendungen: Einführung in Sockets, Java, RMI, CORBA und Jini*. Oldenburg Wissenschaftsverlag GmbH, Oldenburg, 2001. ISBN:978-3-486-58481-3.

[Ill07] J. A. Illik. *Verteilte Systeme;Architekturen und Software-Technologien*. expert Verlag, 2007. ISBN:978-3-8169-2730-3.

[Ilm06] TU Ilmenau. Traceability for managing evolutionary change, 2006. http://www.tu-ilmenau.de/fakia/Traceability-for-Man.6036.0.html; letzter Zugriff:2010.02.01.

[Jav] Sun Developper Network Java. Enterprise javabeans technology. http://java.sun.com/products/ejb/docs.html; letzter Zugriff: 2010.02.06.

[KB07] M. Kuhrmann and G. Beneken. *Widows Communication Foundation Konzepte-Programming-Migration*. Littmann-Bähr, Ungarn, Ungarn, 2007. ISBN:978-3-8274-1598-1.

[KCH04] M. Kuhrmann, J. Calamé, and E. Horn. *Verteilte Systeme mit .Net Remoting*. 2004. ISBN:3-8274-1545-4.

[Ker08] S. Kersken. *IT Handbuch für Informatiker*. Gallileo Press, Bonn, 2008. ISBN:3-446-40185-7.

[Kit08] C. Kittler. Konzeption und prototypische umsetzung einer komponente zur ablage von traceability informationen in enterprise architect modellen. *Studienjahresarbeit*, Juni 2008.

[Kus07] T. Kuschke. Automatische nachführung von traceability links zwischen analyse- und designobjekten im sw-entwicklungsprozess. Master's thesis, TU Ilmenau, 2007.

[MBW08] P. Mandl, A. Bakomenko, and J. Weiß. *Grundkurs Datenkommunikation*. Vieweg+Teubner | GWV Fachverlage GmbH, 2008. ISBN:979-3-8348-0517-1.

[MGP08] P. Mäder, O. Gotel, and Ilka Philippow. Enabling automated traceability maintenance by recognizing development activities applied to models. *Proceedings 23rd International Conference on Automated Software Engineering*, Sep 17-19 2008.

[MGP09a] P. Mäder, O. Gotel, , and I. Philippow. Enabling Automated Traceability Maintenance Through the Upkeep of Traceability Relations. *Proceedings 5th European Conference on Model-Driven Architecture Foundations and Applications*, 2009.

[MGP09b] P. Mäder, O. Gotel, and I. Philippow. Semi-automated traceability maintenance: An architectural overview of tracemaintainer. *Proceedings 5th ECMDA Traceability Workshop*, 2009.

[MNW03] S. McLean, J. Naftel, and K. Williams. *Microsoft .Net Remoting*. Microsoft Press, 2003. ISBN:0-7356-1778-3.

[MRP06a] P. Maeder, M. Riebisch, and I. Philippow. Aufrechterhaltung von traceability links während evolutionärer softwareentwicklung. *Proceedings 8th Workshop Software-Reengineering*, 26(3), May 2006.

[MRP06b] P. Maeder, M. Riebisch, and I. Philippow. Traceability for Managing Evolutionary Change. *Proceedings of 15th International Conference on Software Engineering and Data Engineering*, 2006.

[MRP07] P. Maeder, M. Riebisch, and I. Philippow. Customizing traceability links for the unified process. *Proceedings International Conference on Quality of Software-Architectures*, July 12-13 2007.

[MSDa] MSDN. Com+ (component services). http://msdn.microsoft.com/de-de/library/ms685978%28en-us,VS.85%29.aspx; Letzter Zugriff: 2010.02.06.

[MSDb] MSDN. Einführung in windows-dienstanwendungen. http://msdn.microsoft.com/de-de/library/d56de412%28VS.80%29.aspx; Letzter Zugriff: 2010.02.08.

[MSDc] MSDN. .net framework developer center. http://msdn.microsoft.com/de-de/netframework/default.aspx; letzter Zugriff: 2010.02.06.

[MSDd] MSDN. .net remoting. http://msdn.microsoft.com/de-de/

library/bb979191.aspx; letzter Zugriff: 2010.02.06.

[MSDe] MSDN. Web services und das .net framework. http://msdn.microsoft.com/de-de/library/cc952543.aspx; letzter Zugriff: 2010.02.06.

[MSD05] MSDN. Wcf, 2005. http://msdn.microsoft.com/de-de/netframework/aa663324.aspx; letzter Zugriff:13.12.2009.

[Mue02] G. Muehl. Large-scale content-based publish/subscribe systems. Master's thesis, Technischen Universitaet Darmstadt, 2002.

[Mye00] G. J. Myers. *Methodisches Testen von Programmen*. Oldenburg, 2000. ISBN:3-486-23414-5.

[PR90] N. Parrington and M. Roper. *Softwaretest, Ziele, Anwendungen, Methoden;*. McGraw-Hill, Hamburg; New York, 1990. ISBN:89028-195-8.

[Prö07] U. Pröller. Vereinheitlichte kommunikation durch die wcf, 2007. http://www.tecchannel.de/webtechnik/entwicklung; Letzter Zugriff: 01.02.10.

[Qoo09] Mr. Qoon. Was ist der hub-und spoke-modell?, 2009. http://www.wasista.de/Autos/2009/0225/Was-ist-der-Hub-und-Spoke-Modell.html; letzter Zugriff: 05.02.10.

[Ram02] I. Rammer. *Advanced .Net Remoting*. ai Appress, USA, 2002. ISBN:1-59059-025-2.

[Rot03] H. Rottmann. *C# .Net mit Methode- Professionelle Software entwickeln mit C# und .Net*. Vieweg, 2003. ISBN:3-528-05845-5.

[Sch93] H. S. Schmitt. *Client-Server-Architekturen; Archiekturmodelle für eine neue Informationstechnische Infrastruktur*. PETER LANG Europäische Verlag der Wissenschaften, 1993. ISBN:3-631-47036-3.

[Sch99] B. Schumacher. Proactive flight schedule evaluation at delta air lines. *Proceedings of the I999 Winter Simulation Conference*, 1999.

[Sys98] Sparx Systems. Automation interface of enterprise architect, 1998. http://www.sparxsystems.com.au/resources/developers/autint.html; Letzter Zugriff: 01.02.10.

[Tha00] G. E. Thaller. *Software-Test, Verifikation und Validation*. Heinze Heise, Hannover, 2000. ISBN:3-88229-183-4.

[TS03] A. S. Tanenbaum and V. M. Steen. *Verteilte Systeme. Grundlagen und Paradigmen*. Elsevier GmbH, München, München. 2003. ISBN:3-8274-1545-4.

[Web98] M. Weber. *Verteilte Systeme*. Spektrum Akademischer Verlag, 1998. ISBN:3-8274-0221-2.

[WH06] H.-D. Wuttke and K Henke. *Schaltsysteme - Eine automatenorientierte Einführung*. Pearson Studium, Bafög-Ausgabe, 2006. ISBN:3-8273-7259-3.

[Zei04] A. Zeidler. A distributed publish/subscribe notification service for pervasive environments. Master's thesis, Technischen Universität Darmstadt, 2004.

Abbildungsverzeichnis

Tabellenverzeichnis

Anlagen

A. Klassendiagramme

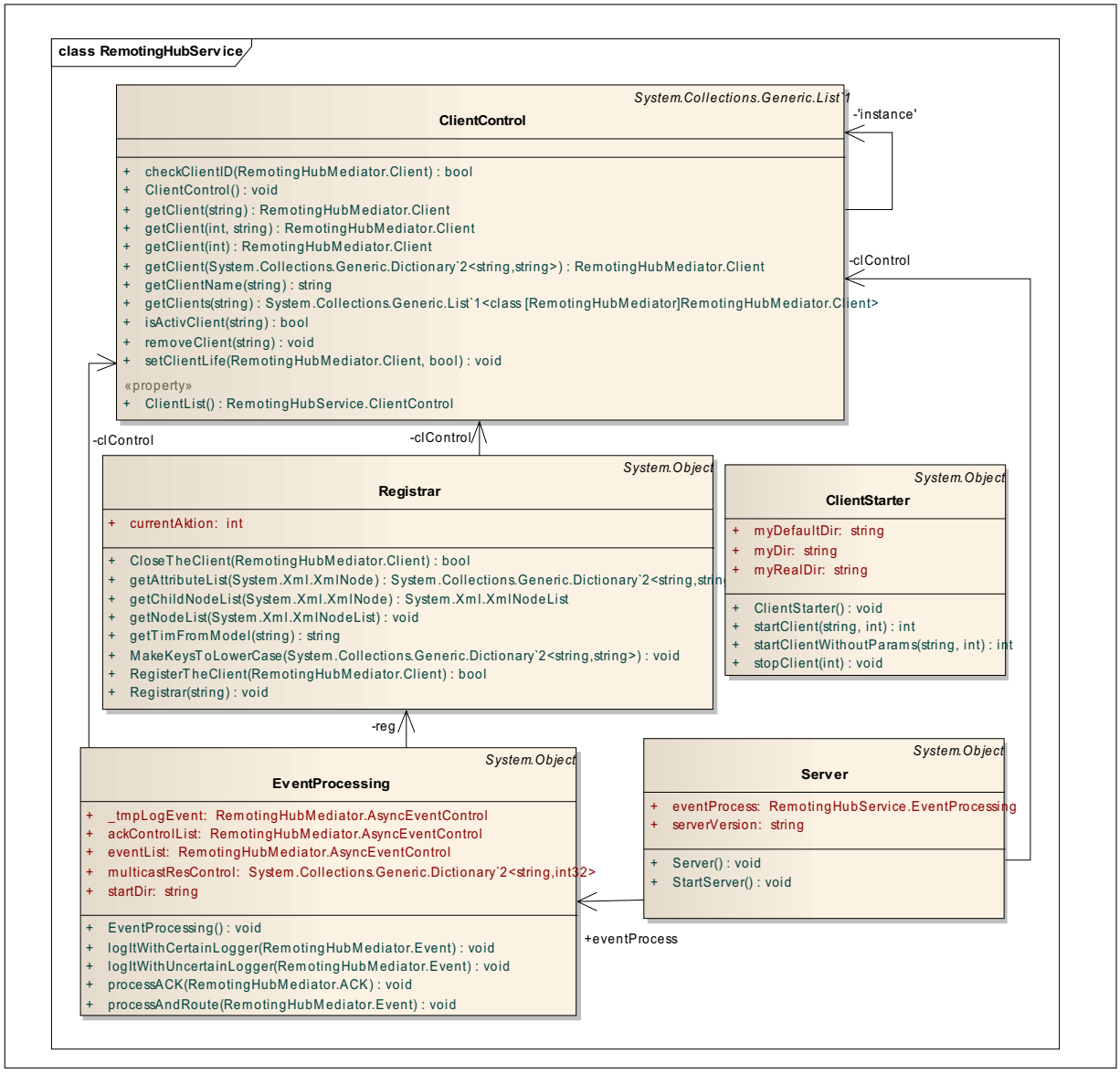

Abbildung A.1.: Klassendiagramm des RemotingHubServices

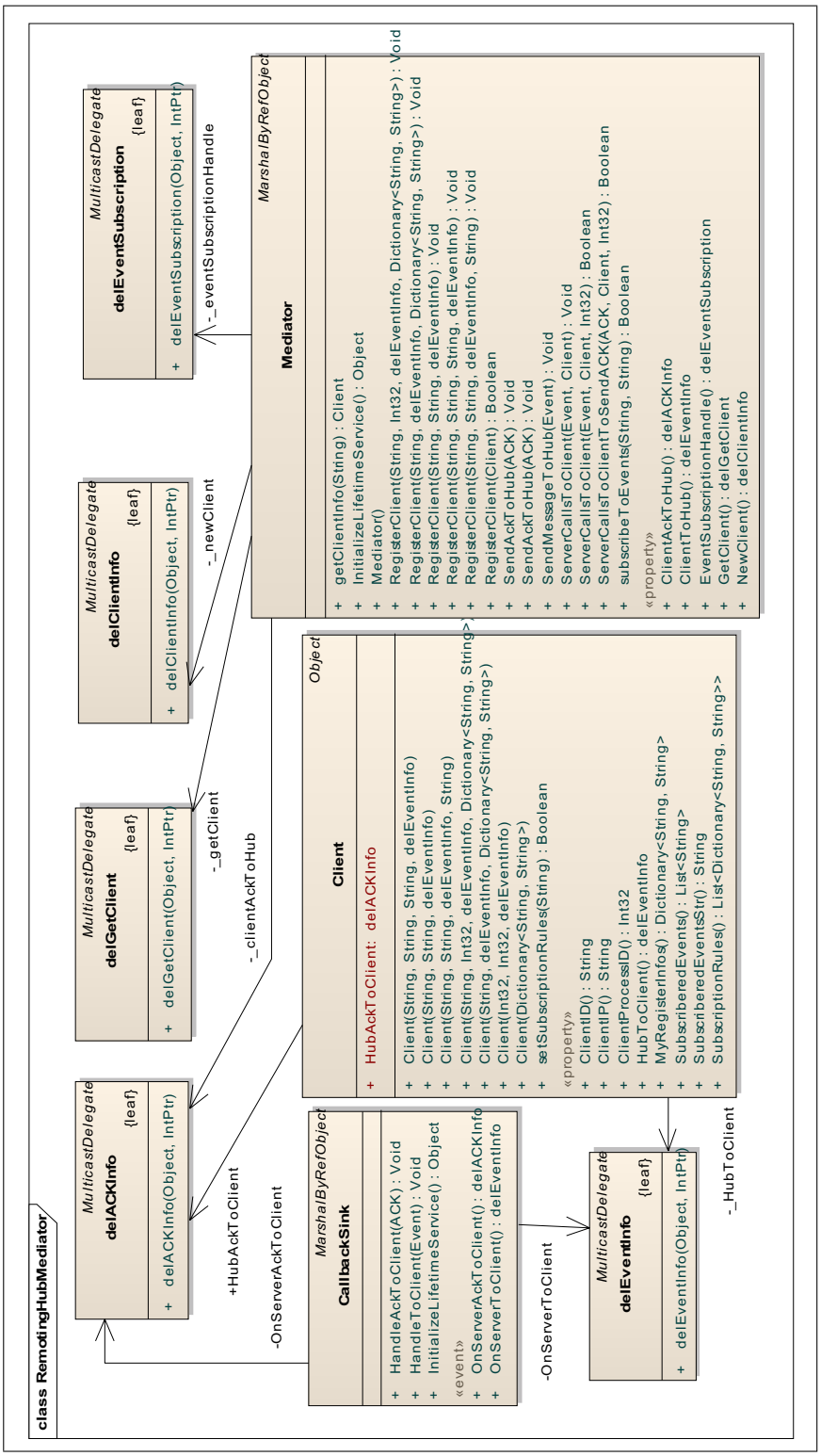

Abbildung A.2.: Klassendiagramm des RemotingHubMediators 1

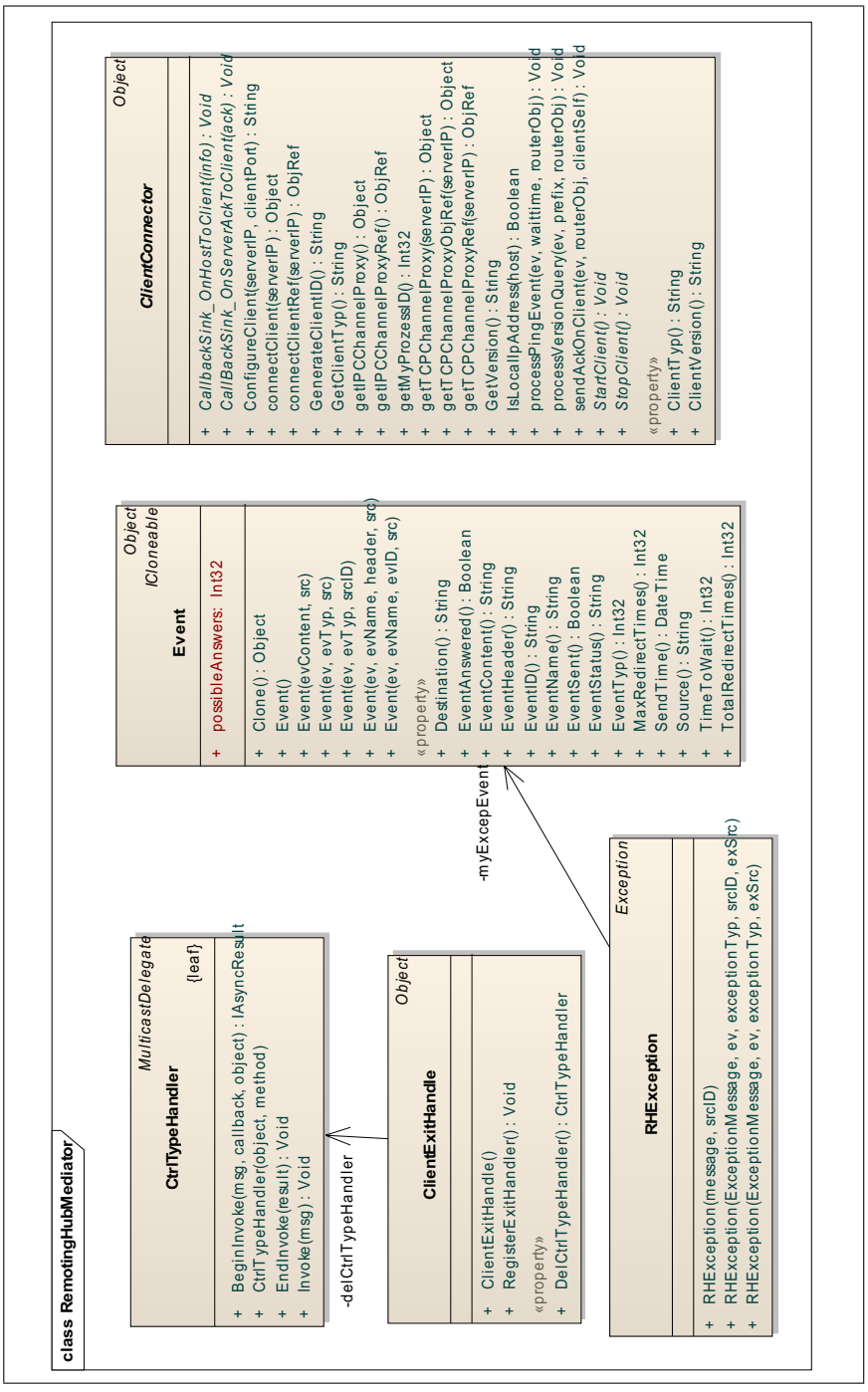

Abbildung A.3.: Klassendiagramm des RemotingHubMediators 2

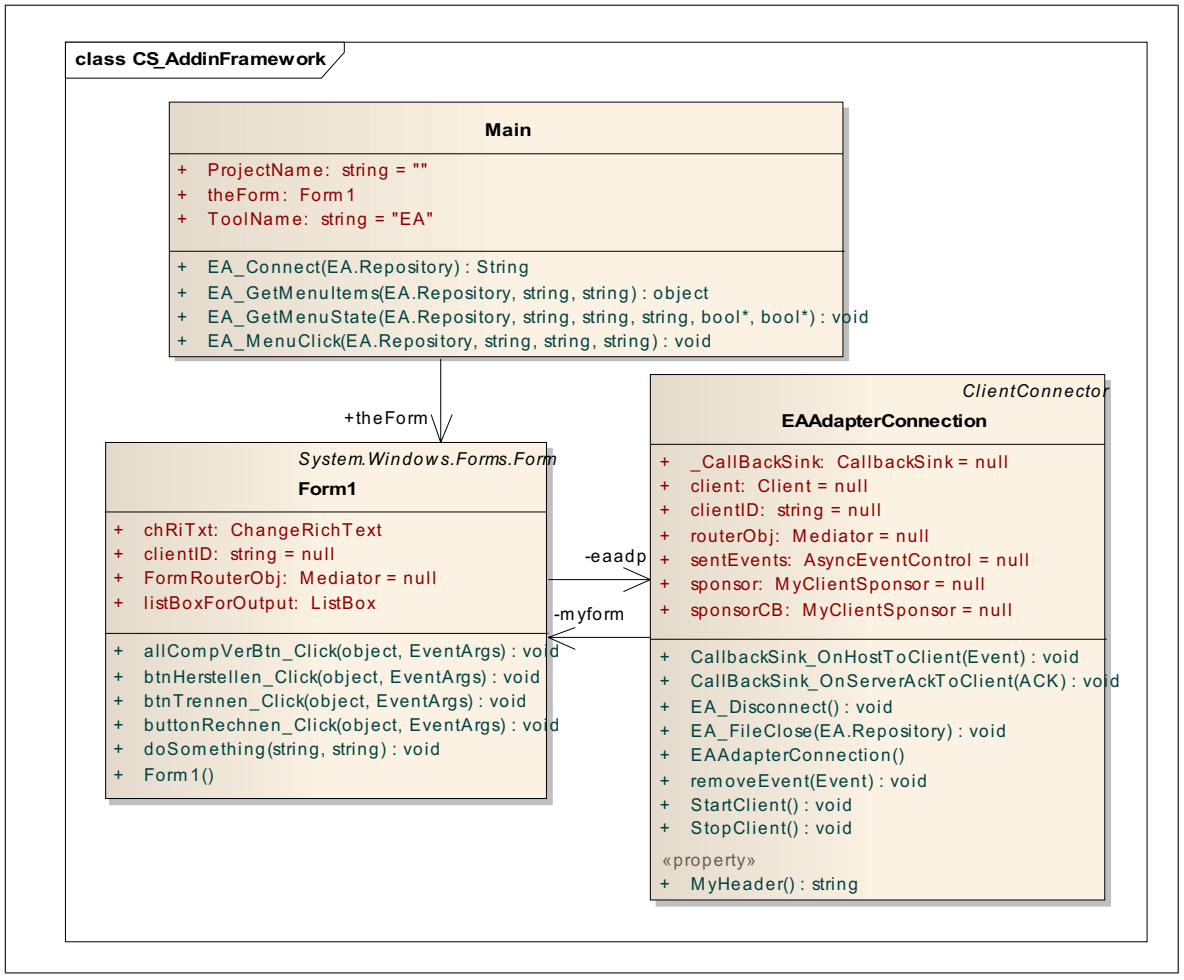

Abbildung A.4.: Klassendiagramm des CSFrameworks

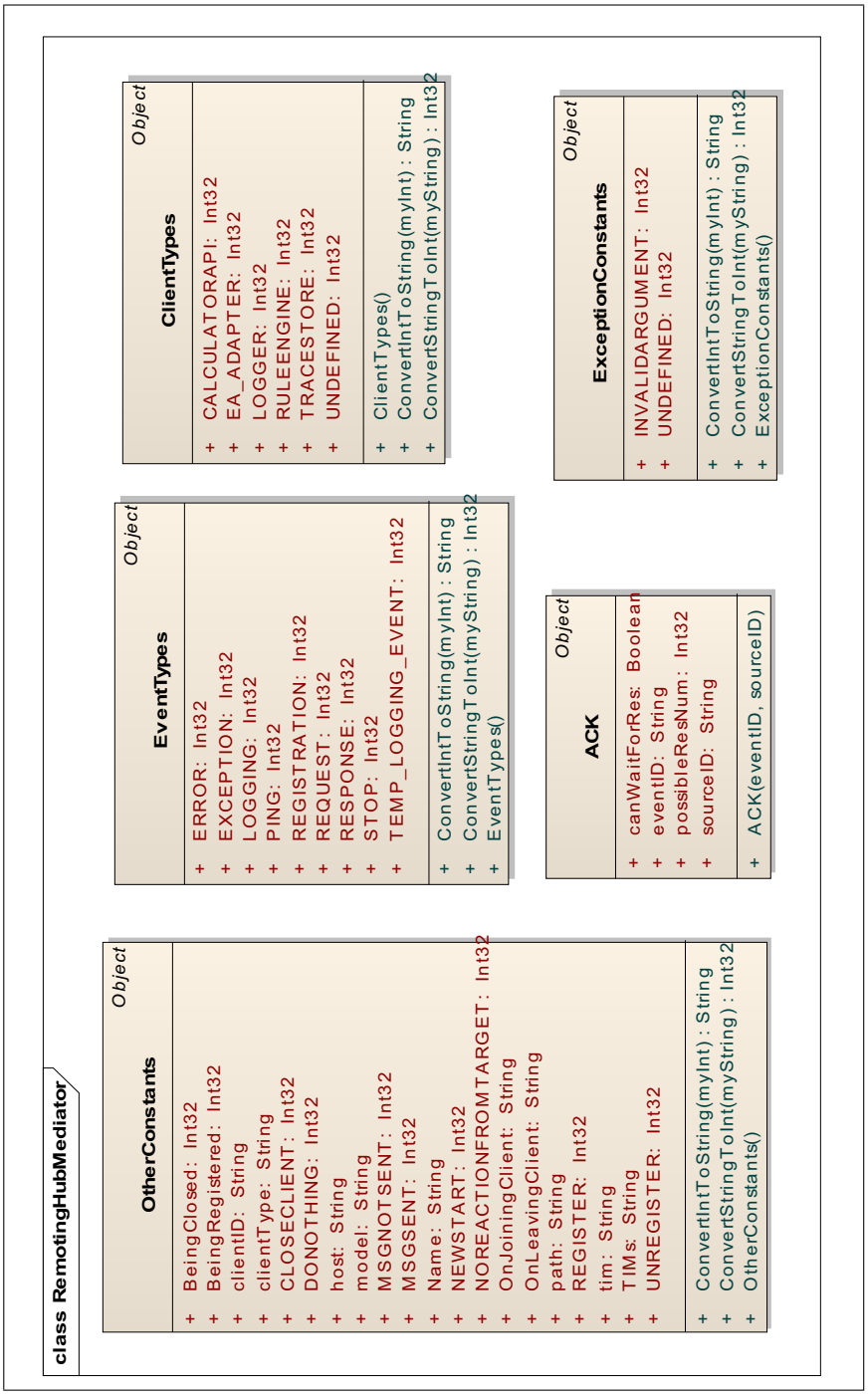

Abbildung A.5.: Klassendiagramm des RemotingHubMediators 3

B. Sequenzdiagramme

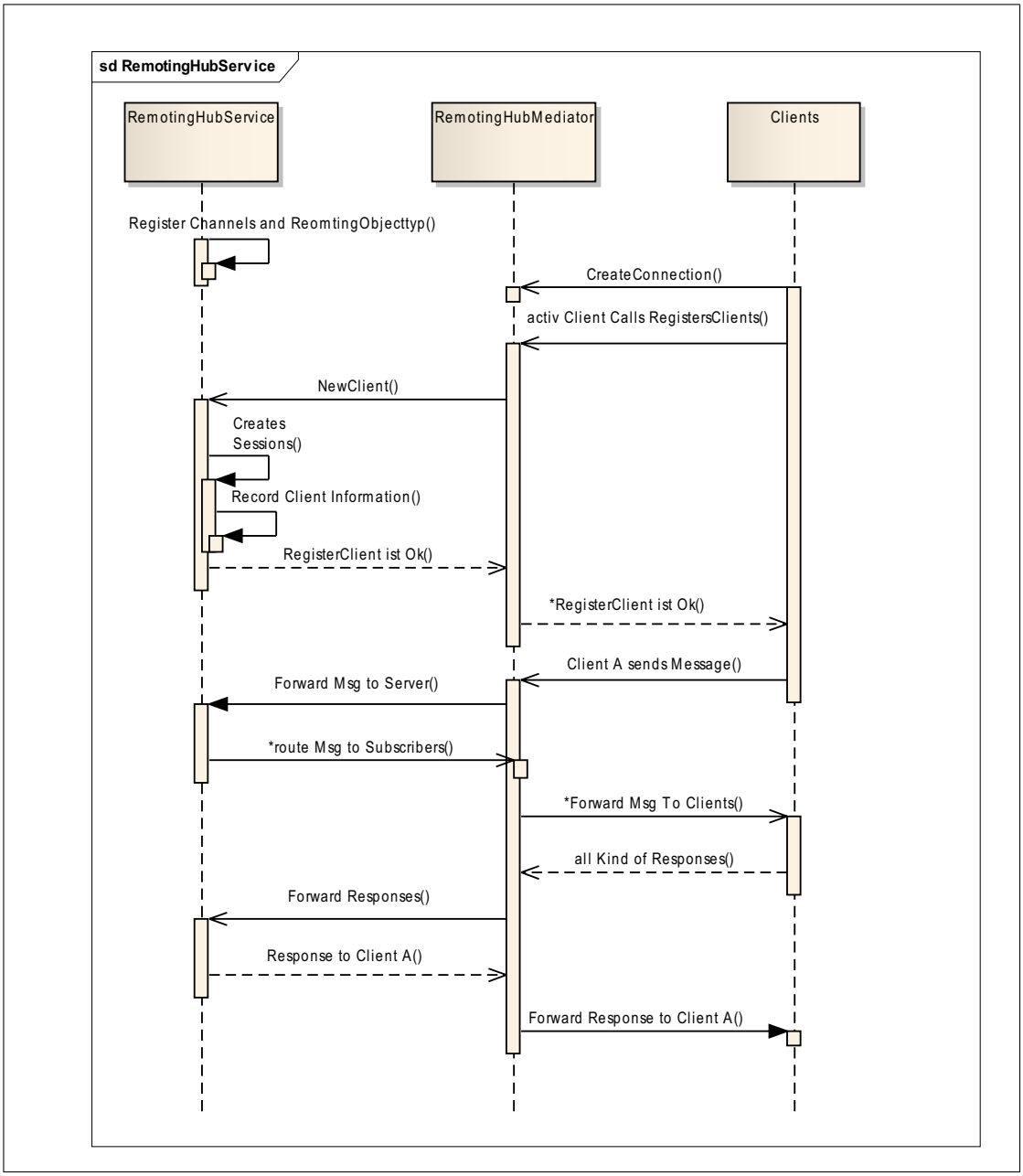

Abbildung B.1.: Sequenzdiagramm des RemotingHubs

C. Abbkürzungen

SOAP: Simple Object Access Protocol
MII: Media Independent Interface
MDI: Mediumabhängige Schnittstelle
MBR: Marshall by Reference
MBV: Marshall by Value
TM: TraceMaintainer
EG: Eventgenerator
TS: TraceStore
EA: Enterprise Architect
UML: Unified Modelling Language
XML: Extensible Markup Language
DNF: Disjunctiv Normal Form
WCF: Windows Communication Foundation
HTTP: Hypertext Transfer Protocol
TCP/IP: Transmission Control Protocol/Internet Protocol
WCF: Windows Communication Foundation
PTP: Peer-to-Peer
CS: Client-Server
SOA: Service oriented Architecture
WSDL: Web Services Description Language
API: Application Programming Interface
UDP: User Datagram Protocol
URI: Uniform Resource Identifier
MSMQ: Message Queuing
CLR: Common Language Runtime
SW: Software
RPC: Remote Procedure Call
LAN: Local Area Network
IIS: Internet Information Service
D/COM: Distributed/Component Object Model
PS-Modell: Publish-Subscribe Model
RMI: Remote Method Invocation
CORBA: Common Object Request Broker Architecture
UDDI: Universal Description, Discovery and Integration
TIM: Traceablitiy Information Model

D. Eidesstattliche Erklärung

Eidesstattliche Erklärung

Hiermit versicher ich, die vorliegende Diplomarbeit selbstständig und nur unter Verwendung der angegebenen Quellen angefertigt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht werden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ilmenau, den 17. Februar 2010